CHAPTER

8 Observing Facts

Although deduction techniques do not take concrete runs into account, observation determines *facts* about what has happened in a concrete run. In this chapter, we look under the hood of the actual program execution and introduce widespread techniques for examining program executions and program states. These techniques include classical logging, interactive debuggers, and postmortem debugging, as well as eye-opening visualization and summarization techniques.

8.1 OBSERVING STATE

Deduction alone, as discussed in Chapter 7 is good for telling what *might* happen. To find out what is *actually* happening in a concrete failing run, though, we cannot rely on deduction alone. We must take a look at the actual facts—that is, *observe* what is going on—and judge whether the values are infected or not. The following are some general principles of observation.

- *Do not interfere*. Whatever you observe should be an effect of the original run—rather than an effect of your observation. Otherwise, you will have a difficult time reasoning about the original run. That is, you have a Heisenbug (see Section 4.3.9 in Chapter 4). Any observation technique should take care to alter the original run as little as possible.
- *Know what and when to observe*. As discussed in Section 1.3 in Chapter 1, a program run is a long succession of huge program states, which is impossible to observe and comprehend as a whole. Observation is effective only if you know:
  - Which part of the state to observe (what)
  - At which moments during execution to observe (when)
- *Proceed systematically*. Rather than observing values at random, let your search be guided by scientific method (Chapter 6). Always be aware of the current hypothesis, the observations predicted by the hypothesis, and how the actual observations contribute to the hypothesis.

In the remainder of this chapter, we shall take a look at some common techniques for observing what is going on in a run. These techniques can be used "as is" by humans, but they can also be leveraged by automated debugging techniques. Here, we ask:

HOW CAN WE OBSERVE A PROGRAM RUN?

8.2 LOGGING EXECUTION

To observe facts about a run, we must make the facts accessible to the programmer. The simplest way of doing so is to have the program output the facts as desired—for instance, by inserting appropriate *logging statements* in the code. For instance, if a programmer wants to know the value of `size` in the function `shell_sort()`, he or she simply inserts a logging statement at the beginning of `shell_sort()`, as follows.

```
printf("size = %d\n", size);
```

Whenever this statement executes, a line such as

```
size = 3
```

will appear on the output device. Several outputs like this constitute a *debugging log*—a list of events that took place during the execution.

This technique of observation is perhaps best known as *printf debugging*—from `printf()`, the function that in C outputs strings and values. (Although C and `printf()` slowly become obsolete, the term *printf debugging* lives on—and JAVA-inspired alternatives such as *system-err-println debugging* are just not catchy enough. Feel free to replace *printf* by the name of your favorite output function.) It is always available in some form. Even if a program might not log on a consolelike device, there always must be some effect that can be observed by the programmer—even if it is just a blinking LED. Being always available (and extremely easy to teach), it is also the most widespread debugging technique. In addition, requiring no infrastructure other than a means of making the log available to the programmer, it is also the most basic technique. Although printf debugging is easy to use, it has several drawbacks as an observation technique.

- *Cluttered code.* Because logging statements serve no other purpose than debugging, they do not help us in understanding the code. The main message a logging statement conveys is that the procedure in question was in need of debugging. Therefore, programmers frequently remove logging statements once the observation is done.
- *Cluttered output.* Logging statements can produce a great deal of output, depending on the number of events and variables traced. If the debugging log is interleaved with the ordinary output, both can be difficult to separate properly. (This problem is best solved by using a designated output channel for debugging logs.)

- *Slowdown.* A huge amount of logging—in particular, to some slow device—can slow down the program. In addition to the obvious performance problem, this changes the program under test and introduces the risk of Heisenbugs (see Section 4.3.9 in Chapter 4).
- *Loss of data.* For performance reasons, output is typically *buffered* before actually being written to the output device. In the event of a program crash, this buffered output is lost. Using ordinary (buffered) output for logging thus hides what happened in the last few events before the crash. One can either use an unbuffered channel for logging (introducing a slowdown, as described previously) or make sure that the program flushes its buffered logs in case of a crash.

Taking care of all of these issues in a single output statement is quite a hassle. Therefore, it is better to use *dedicated logging techniques* that allow far better customization. In particular, we would like to do the following:

- *Use standard formats.* Standard formats make it easy to search and filter logs for:
  - Specific *code locations* (“prefix each line with the current file or function”)
  - Specific *events* (“prefix each line with time”)
  - Specific *data* (“output all dates in Y-M-D format”)
- *Make logging optional.* For performance reasons, logging is typically turned off in production code as well as in code not under consideration for debugging.
- *Allow for variable granularity.* Depending on the problem you are working on, it may be helpful to focus on specific levels of detail. Focusing only on specific events also improves performance.
- *Be persistent.* One should be enabled to reuse or reenable logging even when the debugging session is completed—just in case a similar problem resurfaces.

8.2.1 Logging Functions

The easiest way of customizing logging is to use or design a function that is built for logging purposes only—a *logging function*. For instance, one could introduce a function named `dprintf()` that would behave as `printf()`, but forward its output to a debugging channel (rather than standard output) and allow the output to be turned off. For instance, one could use

```
dprintf("size = %d", size);
```

to output the variable `size` to the debugging log, possibly prefixed with common information such as date or time, or a simple marker indicating the type of output:

```
DEBUG: size = 3
```

In addition, a function such as `dprintf()` can be easily set up not to generate any output at all, which is useful for production code. In practice, though, a programmer would not want to rely exclusively on such a debugging function—particularly if the logging code is to survive the current debugging session. The reason is performance. Even if `dprintf()` does nothing at all, the mere cost of computing the arguments and calling the function may be a penalty if it occurs often.

Languages with a *preprocessor* (such as C and C++) offer a more cost-effective way of realizing logging. The idea is to use a *logging macro*—a piece of code that expands to a logging statement or a no-op (a statement without any effect), depending on settings made at compilation. The following is a simple example of a `LOG()` macro that takes `printf()` arguments in parentheses.

```
LOG(("size = %d", size));
```

The macro `LOG()` is easily defined as being based on `dprintf()` (or `printf()`, or any other suitable logging function).

```
#define LOG(args) dprintf args
```

The effect of this macro definition is that `LOG(args)` is being replaced by `dprintf args` in the subsequent code. Thus, the statement

```
LOG(("size = %d", size));
```

expands into

```
dprintf("size = %d", size);
```

The main benefit of writing `LOG()` rather than `dprintf()` is that a macro can be set up to expand into nothing.

```
#define LOG(args) ((void) 0)
```

Thus, all `LOG()` statements get expanded to a no-op. Not even the arguments will be evaluated. Therefore, `LOG()` statements may even contain calls to some expensive function, as in

```
LOG(("number_of_files = %d", count_files(directory)));
```

If `LOG()` is defined to be a no-op, `count_files()` will not be called—in contrast to the argument to a no-op function.

The choice between turning logging on or off is typically made at compile time. For instance, defining a preprocessor variable (say, `NDEBUG` for "no debugging") during compilation may turn logging off.

```
#if !defined(NDEBUG)
#define LOG(args) dprintf args
#else
#define LOG(args) ((void) 0)
#endif
```

In addition to performance benefits, macros bring a second advantage over functions: They can convey information about *their own location.* In C and C++, the macros `__FILE__` and `__LINE__` expand to the name of the current source file and the current source line, respectively. This can be used in a macro definition such as the following.

```
#define LOG(args) do { \
    dprintf("%s:%d: ", __FILE__, __LINE__); \
    dprintf args; \
    dprintf("\n"); } while (0)
```

(The `do ... while` loop makes the macro body a single statement, for having code such as `if (debug) LOG(var);` work the intended way.) If we insert a `LOG()` macro in line 8 of `sample.c`, its logging output will automatically be prefixed with the location information, as in

```
sample.c:8: size = 3
```

This feature of reporting the location in a macro makes it easy to trace back the log line to its origin (such as `sample.c:8` in the previous example). It can also be leveraged to ***filter*** logs at runtime. For instance, one could set up a function named `do_we_log_this(file)` that returns true if `file` is to be logged (by looking up some configuration resource such as an environment variable). Then, we could introduce a conditional `LOG()` using:

```
#define LOG(args) do { \
    if (do_we_log_this(__FILE__)) { \
        dprintf("%s:%d: ", __FILE__, __LINE__); \
        dprintf args; \
        dprintf("\n"); \
    } } while (0)
```

It is easy to see how these pieces fall into place to produce a set of macros and functions that allow for easy logging of an arbitrary state—using standard formats, with optional logging, and variable granularity. With a bit of discipline, such logging code can even become persistent and remain in the source code. Thus, later programmers can observe the specified events just by turning on some configuration option.

Logging functions are not just useful for making logging optional; they can also help standardize the output of large data structures. Assume we have a very basic linked list, defined as:

```
struct list {
    int elem;             // List element
    struct list *next;    // Next node, or NULL
};
```

We can create a variant of the `LOG()` macro to log the content of a linked list:

```
#define LOG_LIST(list) do { \
    if (do_we_log_this(__FILE__)) { \
        dprintf("%s:%d: ", __FILE__, __LINE__); \
        dprintf("%s = ", #list); \
        print_list(list); \
        dprintf("\n"); \
    } } while (0)
```

In a C macro, the expression #VAR expands to a string containing the macro argument VAR. We use this to log the variable name. If we invoke the macro as LOG_LIST(my_list), then #list becomes "my_list" and the log starts with "my_list = ." The print_list function invoked does a simple traversal of the list, printing its elements:

```
void print_list(struct list *l)
{
    int number_of_elems = 0;
    printf("[");

    while (l != NULL)
    {
        if (++number_of_elems > 1)
            printf(", ");
        printf("%d", l->elem);
        l = l->next;
    }
    printf("]");
}
```

Overall, LOG_LIST(my_list) thus logs something such as:

```
list.c:47: my_list = [1, 10, 100, 1000, 10000]
```

Any large program contains functions to output central data structures in a human-readable form. In C++, such functions typically overload the << operator such that they can write to arbitrary output streams. In JAVA, the standard is to provide a toString() method, which returns a human-readable string for the object.

8.2.2 Logging Frameworks

Although many projects include their own home-grown logging facilities, there are also *standard libraries* for logging, providing a wealth of functionality seldom present in individual projects. As an example, consider the LOG4J framework, a popular logging framework for JAVA programs (also available for C, C++, C#, PERL, PYTHON, RUBY, and EIFFEL).

The core idea of LOG4J is to assign each class in an application an individual or common *logger.* A logger is a component that takes a request for logging and logs it. Each logger has a *level,* from DEBUG (messages of low importance) over INFO, WARN, and ERROR, to FATAL (very important messages). Messages for each of these levels are logged by invoking the corresponding logger methods (debug(), info(), ..., fatal()).

Example 8.1 shows how to use a logger, using the universal UGLI interface. The TestLogging class initializes a logging category, named after the class in which it is instantiated. Then, we can use the logger methods to log individual messages. The TestLogging class, when executed, creates a log starting with:

```
Start of main()
A log message with level set to INFO
A log message with level set to WARN
```

EXAMPLE 8.1: A sample test file using LOG4J

```
import org.apache.ugli.ULogger;
import org.apache.ugli.LoggerFactory;

// How to use log4j
public class TestLogging {

    // Initialize a logger.
    final ULogger logger = LoggerFactory.getLogger(TestLogging.class);

    // Try a few logging methods
    public static void main(String args[]) {
        logger.debug("Start of main()");
        logger.info ("A log message with level set to INFO");
        logger.warn ("A log message with level set to WARN");
        logger.error("A log message with level set to ERROR");
        logger.fatal("A log message with level set to FATAL");

        new TestLogging().init();
    }

    // Try some more logging methods
    public void init() {
        java.util.Properties prop = System.getProperties();
        java.util.Enumeration enum = prop.propertyNames();

        logger.info("*** System Environment As Seen By Java ***");
        logger.debug("*** Format: PROPERTY = VALUE ***");

        while (enum.hasMoreElements()) {
            String key = (String) enum.nextElement();
            logger.info(key + " = " + System.getProperty(key));
        }
    }
}
```

```
     A log message with level set to ERROR
     A log message with level set to FATAL
     Calling init()
     *** System Environment As Seen By Java ***
     *** Format: PROPERTY = VALUE ***
     java.runtime.name = Java(TM) 2 Runtime Environment, Standard
     Edition
     sun.boot.library.path = /System/Library/.../1.4.2/Libraries
     java.vm.version = 1.4.2-38
   ⋮
```

The interesting thing about LOG4J is that one can customize every aspect of the log. In particular, one can define specific *logging levels* for individual classes.

For instance, one can set up LOG4J such that for the application only messages of level `ERROR` and higher are shown—except for a specific class, for which we want all messages of `DEBUG` and higher. Furthermore, one can set up specific *appenders,* which direct the log to a particular output (files, console, database, mail, servers). This can be done in a particular *layout.*

All of this can be defined at runtime using configuration files. The following configuration file defines a specific layout, where *conversion patterns* such as `%d` or `%t` insert the current time of function before the actual message (`%m`).

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c %x - %m%n
```

This configuration pattern changes the format of the layout to:

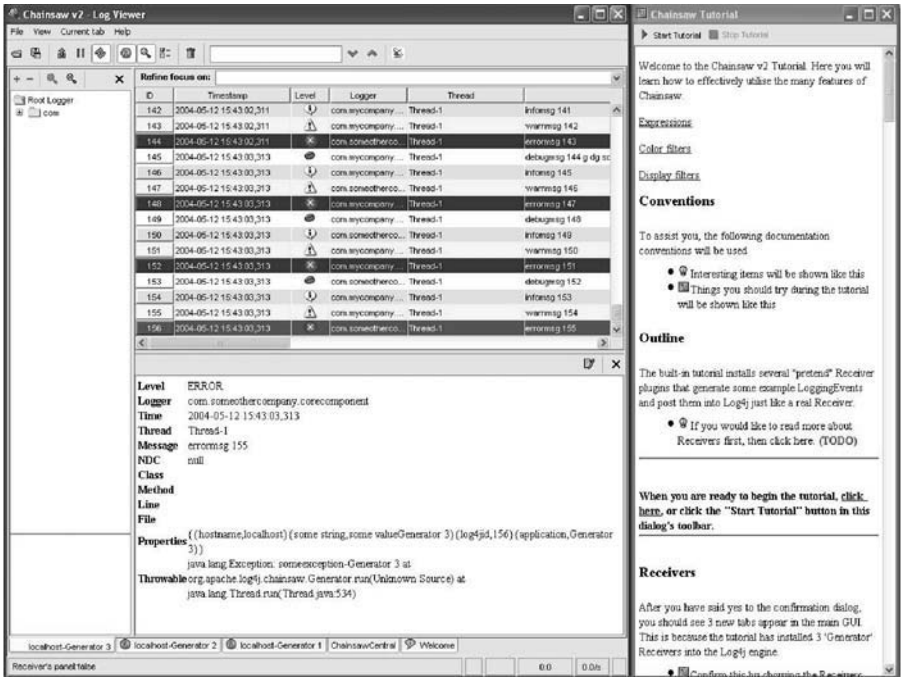
```
    2005-02-06 20:47:31,508 [main] DEBUG  TestLogging - Start of main()
    2005-02-06 20:47:31,529 [main] INFO   TestLogging - A log message
                                            with level set to INFO
.
.
.
```

Because such log files can become painfully long, LOG4J comes with an analysis tool called CHAINSAW that helps to explore these logs. As seen in the screen-shot in Figure 8.1, searching for specific events, levels, or messages is straight-forward.

LOG4J is a very powerful logging package that includes everything but the kitchen sink. Despite its functionality, it is easy to set up initially, and with a little bit of planning, scales up to very large applications. There is every reason to replace `printf()`, `System.out.println()`, and similar output methods with the appropriate `logger` calls from LOG4J and like packages for other languages.

8.2.3 Logging with Aspects

Despite their benefits, logging statements still clutter the source code. The concern of logging is separate from the concern of computation—which is why most programmers prefer to remove logging statements once the debugging session is done. Some languages, though, offer an alternative: Rather than intertwining actual computation and logging they treat these two concerns as separate entities called *aspects*. Each aspect holds only the code for its individual concern. A logging aspect thus holds the logging code in one syntactical entity called an *advice.*



ID	Timestamp	Level	Logger	Thread	
142	2004-05-12 15:43:02,311		com.mycompany....	Thread-1	infomsg 141
143	2004-05-12 15:43:02,311		com.mycompany....	Thread-1	warnmsg 142
144	2004-05-12 15:43:02,311		com.someotherco...	Thread-1	errormsg 143
145	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	debugmsg 144 g dg sc
146	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	infomsg 145
147	2004-05-12 15:43:03,313		com.someotherco...	Thread-1	warnmsg 146
148	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	errormsg 147
149	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	debugmsg 148
150	2004-05-12 15:43:03,313		com.someotherco...	Thread-1	infomsg 149
151	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	warnmsg 150
152	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	errormsg 151
153	2004-05-12 15:43:03,313		com.someotherco...	Thread-1	debugmsg 152
154	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	infomsg 153
155	2004-05-12 15:43:03,313		com.mycompany....	Thread-1	warnmsg 154
156	2004-05-12 15:43:03,313		com.someotherco...	Thread-1	errormsg 155

```
Level      ERROR
Logger     com.someothercompany.corecomponent
Time       2004-05-12 15:43:03,313
Thread     Thread-1
Message    errormsg 155
NDC        null
Class
Method
Line
File
Properties {(hostname,localhost)(some string,some valueGenerator 3)(log4jid,156)(application,Generator
           3)}
           java.lang.Exception: someexception-Generator 3 at
Throwable  org.apache.log4j.chainsaw.Generator.run(Unknown Source) at
           java.lang.Thread.run(Thread.java:534)
```

Welcome to the Chainsaw v2 Tutorial. Here you will learn how to effectively utilise the many features of Chainsaw.

Expressions

Color filters

Display filters

Conventions

To assist you, the following documentation conventions will be used

- Interesting items will be shown like this
- Things you should try during the tutorial will be shown like this

Outline

The built-in tutorial installs several "pretend" Receiver plugins that generate some example LoggingEvents and post them into Log4j just like a real Receiver.

- If you would like to read more about Receivers first, then click here. (TODO)

When you are ready to begin the tutorial, click here, or click the "Start Tutorial" button in this dialog's toolbar.

Receivers

After you have said yes to the confirmation dialog, you should see 3 new tabs appear in the main GUI. This is because the tutorial has installed 3 'Generator' Receivers into the Log4j engine.

FIGURE 8.1

Exploring logs with CHAINSAW. Events can be sorted, filtered, and shown in detail.

The following is a small example. Suppose we have some `Article` class with a `buy()` method.

```
class Article {
   public void buy() {
      // Code
   }
}
```

We want to log the execution of `buy()`, but without actually changing the method. We first write an advice—a simple piece of code logging that `buy()` was called.

```
{
    System.out.println("Calling Article.buy()");
}
```

Alternatively, we do this as a LOG4J afficionado (Section 8.2.2).

```
{
    logger.debug("Calling Article.buy()");
}
```

In the ASPECTJ language, we now have to specify the *location* at which this advice is to be executed. This is done using a *point cut*—a set of locations (*join* *points*) at which the previously cited advice is to be woven into the code. We name this point cut `buyMethod()`.

```
pointcut buyMethod():
    call(public void Article.buy());
```

The advice and the point cut are combined in an *aspect*—for instance, an aspect named `LogBuy`.

```
public aspect LogBuy {
    pointcut buyMethod():
        call(public void Article.buy());
    before(): buyMethod() {
        System.out.println ("Calling Article.buy()")
    }
}
```

The word `before()` means that the advice is to be executed *before* the actual call. We can also specify some advice to be executed after the call has completed.

```
public aspect LogBuy {
    pointcut buyMethod():
        call(public void Article.buy());
    before(): buyMethod() {
        System.out.println ("Entering Article.buy()")
    }
    after(): buyMethod() {
        System.out.println ("Leaving Article.buy()")
    }
}
```

Such an aspect can now be *woven into* the original code, resulting in an executable that is roughly equivalent to:

```
class Article {
   public void buy() {
      System.out.println("Entering Article.buy()");
      original_buy();
      System.out.println("Leaving Article.buy()");
   }
   public void original_buy() {
      // Original code of Article.buy()
   }
}
```

Note, though, that this transformation takes place at the executable level (no source code is ever produced or changed). This weaving is done by the ASPECTJ compiler `ajc`, which substitutes the original JAVA compiler.

```
$ ajc LogBuy.aj Shop.java
$ java Shop
Entering Article.buy()
Leaving Article.buy()
Entering Article.buy()
Leaving Article.buy()
⋮
$ _
```

Weaving in an aspect, though, is *optional*—that is, aspects such as `LogBuy` can also be left away, effectively turning all logging off. Using the ASPECTJ compiler without any aspects is equivalent to using the JAVA compiler alone.

Nonetheless, you may wonder about whether specifying the aspect is worth the hassle. After all, we need a lot of fixture just to insert small advices into the code. The interesting thing about aspects, however, is that the same advice may be woven in at *multiple locations* in the program. For instance, we can specify a point cut that encompasses *all methods* of the `Article` class.

```
pointcut allMethods():
    call(public * Article.*(..));
```

As usual, a star is a wildcard for arbitrary names and qualifiers. Such a point cut can now be used to log multiple methods, all in one place. The variable `thisJoinPoint` can be used to log the name of the current method:

```
public aspect LogArticle {
    pointcut allMethods():
        call(public * Article.*(..));
    before(): allMethods() {
        System.out.println ("Entering " + thisJoinPoint);
    }
    after(): allMethods() {
        System.out.println ("Leaving "  + thisJoinPoint);
    }
}
```

Using wildcards and other pattern expressions for class and method names, such aspects can be easily extended to log an even greater number of methods—or even every single method of the entire program.

Just logging that some method was called is rarely enough. One also wants to log the program state at the event. This is done by integrating the current object and its parameters into the point cut such that they can be accessed in the advice. As an example, imagine we want to log all moves of a `Line` object—that is, all invocations of the `Line.setPX()` and `Line.setPY()` methods. We define a point cut that encompasses these join points and assigns names to the object and the argument. These names can then be used in the advice.

```
public aspect LogMoves {
    pointcut setP(Line a_line, Point p):
        call(void a_line.setP*(p));

    after(Line a_line, Point p): setP(a_line, p) {
        System.out.println(a_line +
                           " moved to " + p + ".");
    }
}
```

These examples should suffice to demonstrate the power of aspects when it comes to observing facts in the program run. Aspects do not clutter the code, and they encourage standard formats, are optional, can be enabled at will, and can easily be reused. The only concern is that logging aspects must not interfere with the actual computation. (Note also that the general idea of having aspects not interfere with each other may also be central for your future aspect-oriented designs.)

8.2.4 Logging at the Binary Level

Aspects, as discussed in Section 8.2.3, require the source code of the program to be logged, and are not available for every language. An alternative is to add logging code not at the source code level but at the binary level—that is, we instrument binary code rather than source code.

The PIN framework provided by Intel is a tool for the instrumentation of Linux binary executables for x86 and other Intel processors. PIN allows arbitrary C or C++ code to be injected at arbitrary places in the executable. It provides a rich API that allows us to access context information such as register content, symbol, and debug information. Conceptionally, you can think of PIN as aspects at the binary level.

In PIN, the actual analysis tools come as so-called PIN tools. They contain the mechanism that decides where and what code to insert (in aspect terminology, a *join point*), as well as the code to be executed at the insertion points (in aspect terminology, the *advice*).

Example 8.2 shows the source code of a simple PIN tool. Running this tool on a binary program creates a trace of all executed instructions—for instance, for the directory listing program `/bin/ls`.

```
$ cd pin-2.0/ManualExamples
$ make itrace
$ ../Bin/pin -t itrace -- /bin/ls
atrace.C  inscount0.C  _insprofiler.C  itrace.o  staticcount.C...
$ _
```

The trace of all instructions is stored in the file `itrace.out`.

```
$ head itrace.out    # output first 10 lines
0x40000c20
0x40000c22
0x40000c70
0x40000c71
0x40000c73
0x40000c74
0x40000c75
0x40000c76
0x40000c79
0x40011d9b
$ _
```

Overall, 501,585 instructions were executed:

```
$ wc -l itrace.out    # count lines in itrace.out
501585
$ _
```

EXAMPLE 8.2: Logging executed instructions

```
// itrace.C - generate an instruction trace

#include <stdio.h>
#include "pin.H"

FILE * trace;

// This function is called before every instruction
// is executed and prints the IP
VOID printip(VOID *ip) { fprintf(trace, "%p\n", ip); }

// Pin calls this function every time
// a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to printip before every instruction,
    // and pass it the IP
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip,
                   IARG_INST_PTR, IARG_END);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace, "#eof\n");
    fclose(trace);
}

// argc, argv are the entire command line,
// including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    trace = fopen("itrace.out", "w");

    // Initialize pin
    PIN_Init(argc, argv);

    // Register Instruction to be called to
    // instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the
    // application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Source: Cohn and Muth, 2004.

How does this work? Let's take a look at the code shown in Example 8.2. The main work is done in three functions:

- The advice to be executed for each instruction is stored in the function `printip()`. The parameter `ip` holds the current instruction address.
- The function `Instruction()` executes every time a new instruction *I* is encountered. It inserts `printip()` as a function to be executed before *I*.
- In `main()`, the `Instruction()` function is registered as a function to instrument instructions.

In addition to operating at the instruction level, the PIN framework also offers means of instrumenting functions (you can retrieve the address of a function, and add advice code to function calls or returns). Thus, it is not too difficult to log a trace of executed functions rather than instructions. Remember the STRACE tool from Section 4.3.6 in Chapter 4, logging the interaction between a program and the operating system? With PIN, you can set up your own logging tool.

8.3 USING DEBUGGERS

The logging techniques discussed in Section 8.2 all require writing and integrating code into the program to be debugged, which takes some time—especially if you consider that the program has to be rerun (and frequently rebuilt) to execute the additional logging code. An alternative mechanism is to use an *external observation tool* that hooks into the execution of the program and observes (and possibly manipulates) the state at specific moments in time—without changing the original program code in any way. This approach has a number of benefits:

- *Getting started fast.* An observation tool can be started right away, without any change to the source code or recompilation.
- *Flexible observation.* It is possible to observe arbitrary aspects of the program execution. Typically, one can even change the aspect during execution.
- *Transient sessions.* Observation tools are good for single shots at debugging, with interaction leading to quick results.

The most important observation tools are known as *debuggers*—not because they actually remove bugs but because they are being used almost exclusively for debugging programs. Debuggers provide three functionalities to help you observe actual executions:

- Execute the program and make it stop on specified conditions
- Observe the state of the stopped program
- Change the state of the stopped program

An example is one of the most powerful debuggers, the GNU debugger (GDB). GDB is an interactive program controlled via a command line. Although your favorite debugger probably comes with a graphical user interface, GDB's command line allows you to focus on the bare debugger functionality—applied on the `sample` program (see Example 1.1).

8.3.1 A Debugging Session

As discussed earlier, the `sample` program is supposed to sort its arguments. However, it has a defect. When invoked with arguments `11` and `14`, the output contains a zero.

```
$ sample 11 14
Output: 0 11
$ _
```

To examine this program run in a debugger, we must first prepare the program for debugging (which seasoned programmers do by default). This means to have the compiler include *debugging information* in the generated executable: locations, names, and types of all variables and functions from the source code. The debugger needs this information in order to find out where a particular item is stored. For GDB, debugging information is included by compiling the program with the `-g` option.

```
$ gcc -g -o sample sample.c
$ _
```

Next, we must load the program into the debugger. (Some debuggers also allow you to attach them to an already-running process.) In the case of GDB, this is done by invoking GDB with the executable as an argument.

```
$ gdb sample
GNU gdb 6.1, Copyright 2004 Free Software Foundation, Inc. ...
(gdb) _
```

The string `(gdb)` is GDB's prompt, at which it accepts a number of commands. At the time of writing, there were 135 different commands built into GDB. However, a few suffice to get acquainted. At first, we must decide *where to stop the program* such that its state can be examined. Following Hypothesis 1 from Section 6.3 in Chapter 6, we first predict that `a[0] = 0` should hold when line 38 is being executed. Therefore, we set a *breakpoint* that will make `sample`'s execution stop at line 38, using the GDB `break` command.

```
(gdb) break 37
Breakpoint 1 at 0x1d04: file sample.c, line 38.
(gdb) _
```

Technically, a breakpoint translates into an *interrupt instruction* that GDB inserts into the executable at the breakpoint location. When execution reaches the breakpoint, the interrupt instruction makes the program stop and returns control to GDB. Now we can actually run the program with the failure-inducing arguments, using GDB's `run` command.

```
(gdb) run 11 14
Starting program: sample 11 14

Breakpoint 1, main (argc=3, argv=0xbffff9f0) at sample.c:38
37          printf("Output: ");
(gdb) _
```

The program has stopped at line 38. Now we can examine the values of individual variables, using GDB's `print` command.

```
(gdb) print a[0]
$1 = 0
(gdb) _
```

GDB reports that the value of `a[0]` is `0`, which confirms the initial hypothesis. (As a courtesy, GDB has saved the printed value in a pseudovariable `$1` such that we can reuse it later—if we run out of zeroes, that is.)

From here, we could now step through the execution, querying variables as we like. GDB provides a `step` and a `next` command that both execute the current line and then stop again. The difference between the two is when the current line is a function call: `step` will go to the first line of the called function, whereas `next` will execute the called function as a whole and remain within the current function.

Instead of stepping through a program without any specific target, it is better to formulate a hypothesis and to verify this hypothesis explicitly. Hypothesis 2 from Section 6.3 in Chapter 6 was that at the beginning of `shell_sort`, `a[] = [11, 14]`, and `size = 2` should hold. The following shows how we can verify this in GDB.

```
(gdb) break shell_sort
Breakpoint 2 at 0x1b00: file sample.c, line 9.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: sample 11 14

Breakpoint 2, shell_sort (a=0x100140, size=3) at sample.c:9
9           int h = 1;
(gdb) print a[0]
$2 = 11
(gdb) p a[1]
$3 = 14
(gdb) p a[2]
$4 = 0
(gdb) _
```

(Note that we can simply type `run` without arguments to reuse the previous arguments, and that we can abbreviate common commands such as `print` to their first letter `p`.) It turns out that `size = 3` holds. Therefore, Hypothesis 2 is rejected.

Hypothesis 3 from Section 6.3 in Chapter 6 states that changing `size` from `3` to `2` should make the run successful. We do so using GDB's `set` command and use `continue` to resume execution.

```
(gdb) set size = 2
(gdb) continue
Continuing.

Breakpoint 1, main (argc=3, argv=0xbffff9f0) at sample.c:38
37          printf("Output: ");
(gdb) _
```

Oops—our first breakpoint is still active. We delete it and finally resume execution, abbreviating `continue` to `c`.

```
(gdb) delete 1
(gdb) c
Continuing.
Output: 11 14

Program exited normally.
(gdb) _
```

Hypothesis 3 is confirmed. We have narrowed down the failure cause to one single variable: `size`. Where does it get its value from? We restart the program and use `where` to show the *backtrace*—the stack of functions that are currently active.

```
(gdb) run
Starting program: sample 11 14

Breakpoint 2, shell_sort (a=0x100140, size=3) at sample.c:9
9           int h = 1;
(gdb) where
#0  shell_sort (a=0x100140, size=3) at sample.c:9
#1  0x00001d04 in main (argc=3, argv=0xbffff9f0) at sample.c:36
(gdb) _
```

It turns out that `shell_sort()` (Frame #0) was invoked by `main()` (Frame #1). To check the local variables of `main()`, we must select its *stack frame*—Frame #1—using the `frame` command.

```
(gdb) frame 1
#1  0x00001d04 in main (argc=3, argv=0xbffff9f0) at sample.c:36
35          shell_sort(a, argc);
(gdb) _
```

This is the place from which `shell_sort()` was invoked, and this is the place we have to fix—by changing `argc` to `argc - 1` (Hypothesis 4 from Section 6.3 in Chapter 6). Many debuggers are incorporated into an editor such that one can change the code on the fly. GDB does not support this, though. Thus, we must fix `sample.c` with an external editor, recompile it, and rerun it.

```
$ sample 11 14
Output: 11 14
$ _
```

Our hypothesis about the failure cause is now refined to a theory. At the same time, we have fixed the defect—all in a five-minute debugger session. We have seen how to:

- Execute the program (`run`) and make it stop on specified conditions (`break`).
- Observe the state of the stopped program (`print`), possibly selecting a frame (`where` and `frame`).
- Resume execution until the next stop (`continue`) or the next line (`next`, `step`).
- Change the state of the stopped program (`set`).

This is the basic debugger functionality, as realized in GDB and almost every other debugger. In the remainder of this section, we discuss other useful functionality.

8.3.2 Controlling Execution

A debugger allows you to control almost every aspect of the environment in which your program executes. This includes:

- Setting of environment and configuration variables:

  ```
  (gdb) set environment USER smith
  (gdb) _
  ```

- Setting of signals and interrupts:

  ```
  (gdb) handle SIGINT ignore        # ignore interrupt
  (gdb) _
  ```

- Hardware-specific settings such as register content:

  ```
  (gdb) set $pc = main     # resume execution at main
  (gdb) _
  ```

8.3.3 Postmortem Debugging

Several operating systems can be set up such that when a program crashes they dump its memory content to a special file. This memory dump (called a *core dump* on UNIX or a *Dr. Watson file* on Windows) can be read in by a debugger such that you may examine the state of the program at the moment it crashed.

The most important hint of a memory dump is the *backtrace,* as it records the functions executing at the time of the crash. Suppose your `fourier` program crashes on a bus error. The default message gives no clue about what might have happened.

```
$ fourier input.txt
Bus error (core dumped)
$ _
```

Loading the memory dump (called `core` on UNIX machines) into the debugger reveals what has happened. Using `where`, we can take a look at the backtrace. `print` reveals the null pointer `a` being dereferenced.

```
$ gdb fourier core
GNU gdb 6.1, Copyright 2004 Free Software Foundation, Inc. ...
Core was generated by './fourier'.
0x00001d8c in init_fourier (a=0x0, x=0) at fourier.c:4
4           a[0] = x;
(gdb) where
#0  0x00001d8c in init_fourier (a=0x0, x=0) at fourier.c:4
#1  0x00001de8 in main (argc=2, argv=0xbffff9e8) at fourier.c:12
(gdb) print a
$1 = (int *)0x0
(gdb) _
```

Even if a program does not leave a memory dump, repeating the run from within the debugger yields the same results—which is why seasoned programmers always test their programs within a debugger.

8.3.4 Logging Data

Some debuggers allow us to execute *commands* automatically. In GDB, for instance, when a breakpoint has been reached it can execute a prerecorded sequence of commands. This can be useful for having breakpoints enable and disable each other, and for realizing *logging of variables* from within the debugger.

In GDB, using the `commands` command we have a breakpoint print the value of a variable (using the GDB `printf` command) and then continue execution. The first command, `silent`, instructs GDB not to report a reached breakpoint.

```
(gdb) break 33
Breakpoint 1 at file sample.c, line 33.
(gdb) commands
Type commands for when breakpoint 1 is hit,
one per line.  End with a line saying just "end".
>silent
>printf "a[%d] = %d\n", i, a[i]
>continue
>end
(gdb) _
```

When executing the program, the value of `i` is logged just as if we had entered an appropriate `printf` command in the source code.

```
(gdb) run
Starting program: sample 7 8 9
a[0] = 7
a[1] = 8
a[2] = 9
...
```

8.3.5 Invoking Functions

Many debuggers allow us to invoke functions of the debugged program. This is frequently used to invoke specific logging functions, as discussed in Section 8.2.1.

```
(gdb) call print_list(my_list)
[1, 10, 100, 1000, 10000]
(gdb) _
```

In GDB, functions can be invoked as parts of expressions, such as:

```
(gdb) print proc.wired_memory() + proc.active_memory()
2578438
(gdb) _
```

Invoking functions can interfere with normal execution of the program. For instance, any side effect of an invoked function affects the program being debugged.

```
(gdb) call clear_list(my_list)
(gdb) call print_list(my_list)
[]
(gdb) _
```

Some side effects are quite confusing, though. For instance, if executing the function reaches a breakpoint, execution will stop at the breakpoint. In such a case, what has happened to the original execution? How do we finish execution of our own invocation, and how do we resume the original execution? Even worse: What happens if the invoked function causes a crash? How do we ensure the original program is not affected? Because of such potential problems, it is wise to invoke only simple functions that do not interfere with the original execution.

8.3.6 Fix and Continue

Some debuggers, integrated within a development environment, allow you to alter the code of the program while it is executing. This way, you can verify whether a fix is successful without having to alter the state explicitly, and without resuming execution from scratch. Be aware, though, that such fixes should be limited to very simple changes. Anything greater creates a mismatch between source code and executable.

8.3.7 Embedded Debuggers

Traditionally, a debugger invokes the program to be debugged. However, one may also set up a system such that it invokes a debugger (or similar interactive facility) on itself. In an interpreted language such as PYTHON, for instance, you can have a program invoke the interactive interpreter, which allows you to explore all of the program's state at will. The following is a very simple piece of code that invokes the interpreter in the middle of a loop by invoking the PYTHON `code.interact()` function with the local scope.

```
import code

for i in range(1, 10):
    print i,
    if i == 5:
        print
        code.interact("Mini Debugger - use Ctrl-D to exit",
                      None, locals())
```

If you execute this code, you obtain:

```
$ python embedded.py
1 2 3 4 5
Mini Debugger - use Ctrl-D to exit
>>>X _
```

In the interpreter, you can enter arbitrary expressions, which are then evaluated. You can also invoke and evaluate arbitrary functions.

```
>>> print i
5
>>> import math
>>> math.sqrt(4)
2.0
>>> _
```

Note, though, that changes to local variables may not affect the (cached) instances in the remainder of the execution. Thus, if you enter `i = 1`, the `i` in the main loop may remain unchanged. Once you are done exploring the state, leaving the interpreter will resume execution of the program.

```
>>> (Ctrl-D)
6 7 8 9
$ _
```

Such an embedded interactive debugging facility can be triggered by inserting appropriate calls in the code, by enabling it from the outside, or upon specific failure conditions. Be aware, though, that this facility is not enabled in production code. Otherwise, bad guys will have fun gaining complete control over your system.

8.3.8 Debugger Caveats

Despite the functionality provided by a debugger, one should keep in mind that interactive debuggers have a certain toylike quality. That is, it is simply fascinating for the creator to see his or her program in action and to exercise total control. This can easily distract from solving the problem at hand. Even when working with an interactive debugger, one should always be explicit about the current hypothesis, and the steps required to validate it, as described in Chapter 6. Debuggers can be excellent tools—but only when combined with good thinking.

8.4 QUERYING EVENTS

Most hypotheses about a program can be tied to a specific *location* within the program, as in "at line 38, a[0] = 0 should hold." This location is the place at which logging code can be inserted, at which aspects can be woven in, and at which a debugger user sets a breakpoint to stop execution.

However, some hypotheses cannot be attached to such a precise location. In fact, the location may well be the subject of a query itself. Just imagine you find some variable (say, Printer.errno) being set to a specific value at the end of a program run. You could now follow back Printer.errno's dependences, as discussed in Chapter 7, and observe each of the locations in which Printer.errno may be set. You will find, though, that in the presence of pointers there will probably be several such locations, and checking them all is a tedious activity. What one needs in this situation is a means of having the program stop at a location that is implied by a condition (e.g., "the location at which Printer.errno is set").

Using an aspect (Section 8.2.3), this is a fairly easy task. All one needs to do is to define a point cut set(Printer.errno) that includes all locations in which the Printer.errno is set.

```
public aspect LogErrno {
  pointcut setErrno():
    set(Printer.errno);

  before(): setErrno() {
    System.out.println("Old value:" + Printer.errno);
  }
  after(): setErrno() {
    System.out.println("New value:" + Printer.errno);
  }
}
```

It is fairly easy to refine this aspect further—for instance, to log Printer.errno only if it gets a specific value.

8.4.1 Watchpoints

Using languages without aspect support such as C, though, we must resort to *debuggers* to catch assignments to specific variables. GDB provides a feature called *data breakpoints* or *watchpoints* to catch the specific moment in time in which a given condition comes true. For instance, in the sample program to catch the moment in which a[0] is assigned use:

```
(gdb) watch a[0]
Watchpoint 3: a[0]
(gdb) _
```

Having created this watchpoint (which must be done when a[0] actually exists), GDB will check for each subsequent machine instruction whether the value of a[0] has changed. If so, GDB will stop program execution.

```
(gdb) continue
Watchpoint 3: a[0]

Old value = 11
New value = 0
shell_sort (a=0x100140, size=3) at sample.c:15
15              for (i = h; i < size; i++)
(gdb) _
```

Execution has stopped at the statement *after* the value of `a[0]` has changed. The most recent executed statement is at the end of the `for` loop, `a[j] = v`, this is where `a[0]` got its zero value from.

Watchpoints are expensive. Because the debugger must verify the value of the watched expression after each instruction, a watchpoint implies a switch between the debugged processes and the debugger process for each instruction step. This slows down program execution by a factor of 1,000 or more. Fortunately, some processors provide *hardware watchpoints* that can automatically monitor a small number of locations for changes without degrading performance. Debuggers such as GDB can make use of such hardware facilities to provide a limited number of hardware-assisted watchpoints.

If your program restricts access to specific data via accessor functions [such as `setX()` and `getX()` methods in a class], it is much easier to set up a breakpoint in these accessor functions. Languages with managed memory, such as JAVA, ensure that no external piece of the program can modify or access data without going through the public interface of the encapsulating class. In languages with unmanaged memory, such as C or C++, protection is limited. It is conceivable that an external piece of the program accesses data directly—either on purpose or by accident (say, via a stray pointer). To catch the culprit, watchpoints can come in as a last resort.

In some cases, one might be interested in having a program stop on a specific condition *and* at a specific location. In this case, a *conditional* breakpoint comes in handy—a breakpoint that stops only under certain conditions. The following is a simple example.

```
(gdb) break 37 if a[0] == 0
Breakpoint 4 at 0x1d04: file sample.c, line 37.
(gdb) _
```

Here, the breakpoint stops the program only if the given condition is met. This is useful when checking for specific infections, such as `a[0]` being zero, in this example.

8.4.2 Uniform Event Queries

The distinction among watchpoints, breakpoints, and conditional breakpoints is purely motivated by the technique by which these concepts are implemented. One might as well *generalize* these concepts to have *a uniform query mechanism* that stops program execution as soon as a specific condition is met—a condition that may involve the current execution position as arbitrary aspects of the program state.

Table 8.1 Attributes in COCA

Events		Data	
Attribute	**Meaning**	**Attribute**	**Meaning**
type	function/return ...	name	Variable name
port	enter/exit	type	Type
func	Function name	val	Value
chrono	Time stamp	addr	Address
cdepth	Call stack	size	Size in memory
line	Current line	linedecl	Declaration line
file	Current file	filedecl	Declaration file

One such attempt was realized by the COCA debugger—a front end to GDB that provides a uniform query for arbitrary events during execution.

In COCA, events and data are characterized by *attributes* (outlined in Table 8.1). These attributes can be used in *queries* to obtain all events or data where the query would be satisfied. A query typically consists of two parts:

- *Time:* A query `fget(`*attributes*`)` denotes the subset of the execution time in which the given attributes hold. The query `fget(func=shell_sort)`, for instance, denotes all events in which the current function is `shell_sort`.
- *Space:* A query `current_data(`*attributes*`)` denotes the subset of the execution data in which the given attributes hold. For example, a query `current_data(type=int)` denotes all data of which the type is `int`.

If time is not specified, the query refers to the data at the current execution position.

Within the specification of time and space, *logical variables* (starting with an uppercase letter) can be used to *match* specific events. Any events that match will then be returned as a result of the query. (Readers familiar with PROLOG may recognize this style of query.) Thus, a full query at COCA's prompt might look as follows.

Which variable currently has a value of `42`**?**

```
[coca] current_var(Name, val=42).
Name = x0
Name = x1
[coca] _
```

Which variables are zero during execution of `shell_sort()`**?**

```
[coca] fget(func=shell_sort and line=Ln),
       current_var(Name, val=0).
Name = a[2]  Ln = ⟨int i, j;⟩
Name = v     Ln = ⟨int v = a[i]⟩
Name = a[0]  Ln = ⟨a[j] = v⟩
[coca] _
```

When did a[2] become zero?

```
[coca] fget(line=Ln),
       current_var(a, val=array(-,-,0,...)).
Ln = ⟨a = malloc(...)⟩
[coca] _
```

Internally, COCA translates all of these queries into appropriate GDB commands:

- `fget()` sets appropriate breakpoints and executes the program.
- `current_var()` queries the set of variables.

Although a single query can generate several GDB commands, the general performance issues of data queries persist. A general query such as

```
fget(func=shell_sort), current_var(a[0], val=0)
```

still requires a watchpoint to capture the moments in which `a[0]` was zero.

Overall, event and data queries as realized in COCA provide a far more general (and far more versatile) interface than the basic debugger breakpoint and watchpoint commands. One should keep in mind, though, that queries over a wide range of events may take a surprisingly long time to execute, because they must rely on slow watchpoints—a surprise that does not take place when programmers must "manually" translate their experiment into appropriate breakpoint and watchpoint commands.

8.5 HOOKING INTO THE INTERPRETER

If one programs in a language that is *interpreted* (rather than compiled), one can easily *hook into the interpreter* to monitor the execution for specific conditions. Such hooks, provided for the implementation of debuggers, can also be used to implement arbitrary observation techniques.

In PYTHON, for instance, the embedded debugger (Section 8.3.7) is built on top of such a hook. The PYTHON function `sys.settrace(tracer)` defines a *tracing function* `tracer` that from now on will be called after every single line of execution. It takes three arguments:

The frame. This is the stack frame of the currently executing function. The frame object provides access to

- the currently executed code (which in turn gives access to properties like function name or file name),
- the current line number, as well as
- the local variables on the stack (which can be read and written).

The event can be `call` (a function is called), `line` (a line is executed), or `return` (a function returns).

The argument is the returned value in case of `return` events.

The `tracer` function returns the next tracer function to be executed (typically itself), or `None`, if tracing is to be turned off.

Here is an example of a PYTHON `tracer` function that outputs a program trace.

```
def tracer(frame, event, arg):
    code = frame.f_code
    function = code.co_name
    filename = code.co_filename
    line = frame.f_lineno
    print filename + ":" + ‘line‘ + ":" + function + "():", \
        event, arg
    return tracer
```

If this tracer is activated by calling

```
sys.settrace(tracer)
```

at the beginning of the program, it generates a trace like

```
./middle.py:13:middle(): call None
./middle.py:14:middle(): line None
./middle.py:15:middle(): line None
./middle.py:16:middle(): line None
./middle.py:18:middle(): line None
./middle.py:19:middle(): line None
./middle.py:26:middle(): line None
./middle.py:26:middle(): return 3
```

Such hooks are available in all interpreted languages that offer interactive debugging facilities. The Java virtual machine (JVM), for instance, has an interface for *monitoring* the access of object attributes. It can interrupt the execution whenever some specific attribute is read or written. Because the JVM is realized in software, there are no limits on the number of monitors, and the performance loss is not as dramatic compared with debugger watchpoints. The `java.lang.instrumentation` package provides so-called *Java agents* who can observe and hook into arbitrary aspects of the execution; so-called *native agents* can even access the internal data structures as used by native methods.

Hooks into the interpreter are the base on which all kinds of observation and diagnostic tools can be built. If you ever wish to implement your own technique, it is a good idea to first experiment with interpreted languages, since observing and altering executions is so much easier. Who knows—eventually, you might be end up writing your own debugger!

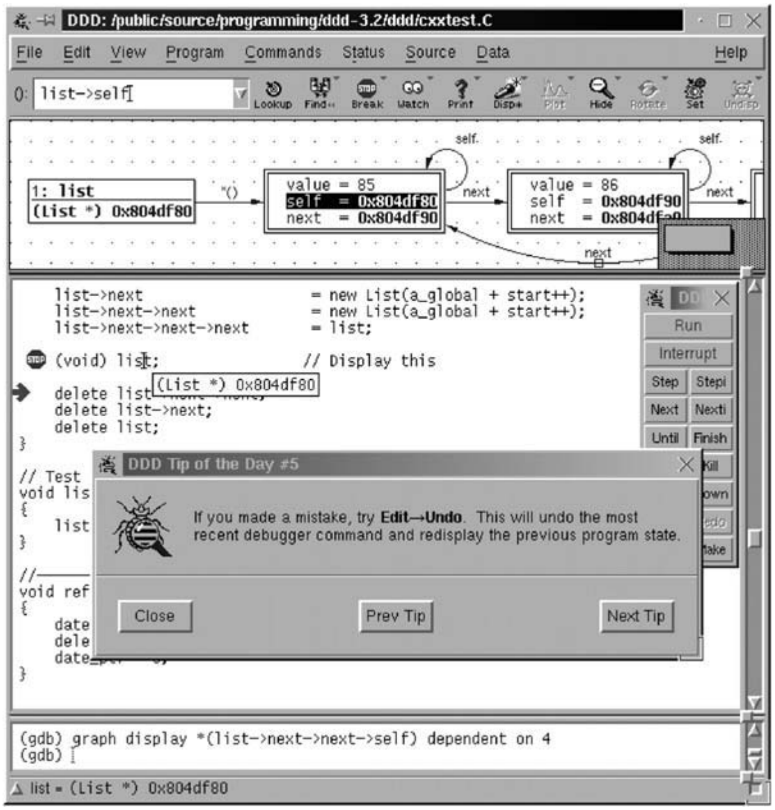
8.6 VISUALIZING STATE

To close this chapter, let me present a more *visual* touch to observation. The techniques discussed so far all have relied on pure *textual* output of program states and variables. The following is an example of a tree node as it is being output by GDB and other debuggers.

```
*tree = {value = 7, _name = 0x8049e88 "Ada",
  _left = 0x804d7d8, _right = 0x0,
  left_thread = false, right_thread = false,
  date = {day_of_week = Thu, day = 1, month = 1,
    year = 1970,
    _vptr. = 0x8049f78 <Date virtual table>},
  static shared = 4711}
```

Although textual output is useful for keeping track of scalar values, it is hardly useful when it comes to tracking *relationships* between objects—especially inclusions and references. In the previous structure, which are the elements of the `date` substructure, for instance? Would you spot whether two pointers have the same address and thus reference the same object?

To improve the understanding of data structures, some debuggers allow us to *visualize* these relationships. The GNU Data Display Debugger, or DDD for short, is a graphical front end for GDB and other debuggers that provides such a visualization (Figure 8.2).


DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
0: list->self
Lookup Find Break Watch Print Disp Plot Hide Rotate Set Undisp
self.
next
Run Interrupt Step Stepi Next Nexti Until Finish Kill Down Undo Make
DDD Tip of the Day #5
Close Prev Tip Next Tip

```
1: list
(List *) 0x804df80

value = 85
self  = 0x804df80
next  = 0x804df90

value = 86
self  = 0x804df90
next  = 0x804df
```

```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;                // Display this

    delete list  (List *) 0x804df80
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//———
void ref
{
    date
    dele
    date_
}
```

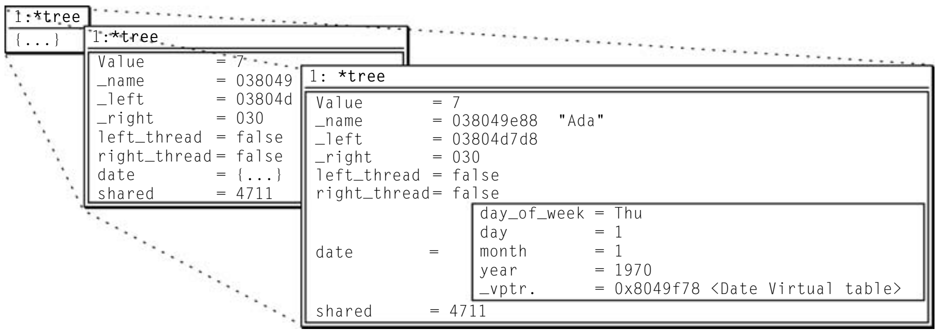
If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

FIGURE 8.2

The DDD debugger. In the top window, DDD has visualized a linked list.



```
1:*tree
{...}

1:*tree
Value        = 7
_name        = 038049
_left        = 03804d
_right       = 030
left_thread  = false
right_thread = false
date         = {...}
shared       = 4711

1: *tree
Value        = 7
_name        = 038049e88  "Ada"
_left        = 03804d7d8
_right       = 030
left_thread  = false
right_thread = false
                 day_of_week = Thu
                 day         = 1
date         =   month       = 1
                 year        = 1970
                 _vptr.      = 0x8049f78 <Date Virtual table>
shared       = 4711
```

FIGURE 8.3

Unfolding data structures in DDD.

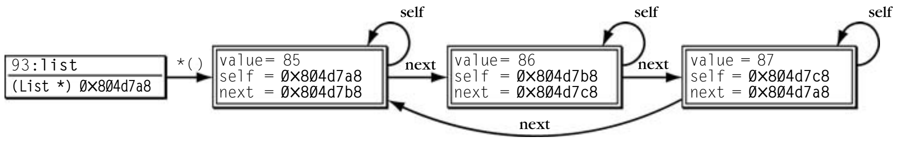


FIGURE 8.4

A linked list in DDD.

DDD visualizes such data as a box with elements that can be unfolded at the user's will (Figure 8.3). This allows the user to focus on specific parts of the data without cluttering the screen with irrelevant data.

DDD displays each individual datum as a single box. However, if a box originates from another box, DDD creates an arrow from the origin to the target. This normally happens if the user dereferences a pointer. A pointer arc points from the origin (the pointer address) to the target (the dereferenced element). This allows for simple exploration and visualization of complex data structures (Figure 8.4).

In addition to visualizing data structures and relationships, DDD can plot numerical values as charts, and even plot the history of individual variables. Such plots summarize multiple values into a single visualization, and make it easier for programmers to detect an uncommon state.

All of these visualizations are limited, though, by the programmer's ability to comprehend and compare large amounts of data. If one already knows the properties of some infection (such as "there is a cycle somewhere in the tree"), it can be easier to have the computer search for these properties rather than scanning huge amounts of data manually. We will come back to this idea in Chapter 10.

8.7 CONCEPTS

When observing state, do not interfere. Know what and when to observe, and proceed systematically.

How To

To observe state, use:

- Logging functions (Section 8.2.1)
- Aspects (Section 8.2.3)
- A debugger (Section 8.3)

Logging statements ("printf debugging") are easy to use, but tend to clutter the code and the output.

To encapsulate and reuse debugging code, use a dedicated logging framework or aspects.

Dedicated logging functions can be set up such that they can be turned off without impacting performance. With enough discipline, they can even remain in the production code.

Aspects offer an elegant way of keeping all logging code in one place and applying the same logging code to multiple places.

Debuggers allow flexible and quick observation of arbitrary events. Reuse of logging code is difficult, though.

To observe the final state of a crashing program, use a debugger to observe the postmortem memory dump. If that is not available, repeat the run in a debugger.

Advanced debuggers allow flexible querying of events (Section 8.4) and visualization of program data (Section 8.6).

To automate observation, hook into the interpreter to gain access to the entire program state.

8.8 TOOLS

LOG4J. The development of LOG4J started in 1996, and has seen countless enhancements and incarnations before it became the popular package it is today. Everything about LOG4J, as well as its ports to other languages, can be found at *http://logging.apache.org/*.

ASPECTJ. ASPECTJ was introduced by Kiczales et al. (2001). Its Web page has several resources to aspect-oriented programming. It can be found at *http://www.eclipse.org/aspectj/*.

PIN. The PIN tool for dynamic binary instrumentation is available at *http://rogue.colorado.edu/Pin/*. The site also contains online manuals and documentation.

BCEL. For JAVA, binary instrumentation is available through BCEL, the Byte Code Engineering Library. It allows arbitrary manipulation of JAVA byte code, including inserting code to be executed before and after function calls. It can be found at *http://jakarta.apache.org/bcel/*.

GDB GDB was developed by Stallman and Pesch (1994), mimicking Sun's DBX interactive debugger. Its Web page is found at *http://www.gnu.org/software/gdb/*.

DDD. DDD was built by Zeller and Lütkehaus (1996) as a front end to GDB Since then, it has been extended to various other command-line debuggers. The DDD manual was written by Zeller (2000), and is available from its Web page at *http://www.gnu.org/software/ddd/*.

JAVA SPIDER. Although DDD also supports JDB, the JAVA command-line debugger, I would not recommend it for debugging JAVA programs. If you are interested in visualizing the JAVA state, have a look at the JAVA SPIDER tool by Erich Gamma and Kent Beck, which can be found at *http://sourceforge.net/projects/ javaspider/*. JAVA SPIDER is publicly available as a plug-in for the ECLIPSE programming environment.

eDOBS. The eDOBS project by Geiger and Zündorf (2002) uses UML diagrams for visualization. It thus raises the level of abstraction from plain programming structures to the level of UML object diagrams. This is especially useful in large-scale program understanding. It can be found at *http://www.se.eecs.uni-kassel.de/se/index.php?edobs*. eDOBS also comes as an ECLIPSE plug-in.

8.9 FURTHER READING

To learn how debuggers such as GDB work, the book by Rosenberg (1996) gives an insight about the basic algorithms, data structures, and architecture of interactive debuggers.

GDB allows efficient watchpoints only with hardware support. As Wahbe (1992) points out, efficient watchpoints need not necessarily be implemented in hardware, and suggests a software solution. His technique modifies the code of the debuggee to monitor the instructions that might affect the watched data—with acceptable performance for most debugging applications.

COCA was developed by Ducassé (1999). An efficient querying concept for JAVA, using JAVA class instrumentation, is described in Lencevicius (2000).

EXERCISES

8.1 Use DDD to debug `sample` (see Section 1.1 in Chapter 1) as follows:

1. Set breakpoints at lines 31, 35, and 37 by pressing the right button of your mouse in these lines at the left border of the source window and selecting "Set Breakpoint" from the resultant context menu.
2. Run the program by selecting "Program → Run" from the menu. Insert your failure-producing arguments and select "Run."
3. The program should have stopped at line 31. Display the content of array `a`: In the argument field, insert `a[0]@5`, and click on the Display button. (Replace 5 with an appropriate number of array fields.) Display `argv` in the same way.

To obtain a display of variables `i` and `argc`, it may be easier to hover with the mouse pointer above a variable name in the source window and use the context menu obtained with the right mouse button.

4. Select “Program → Continue” (or click on the Cont button on the small navigation window).
5. The program should have stopped at line 35. Click on the Continue button.
6. The program should have stopped at line 37. Click on the Continue button.
7. The program should run to the end. To restart the program, you can simply click on “Run again” (or on “Run” in the small navigation window).

Inspect the content of the variables at each breakpoint. Which variables have changed? When does the state become infected?

8.2 Insert logging macros into the `sample` program (see Section 1.1 in Chapter 1). Make your logging macros optional at compile time. Can you determine the behavior of the `shell_sort()` function with the log output only?

8.3 In the Chapter 5 exercises, we used a JAVA implementation of the delta debugging algorithm (Example 5.10) to simplify inputs. In this exercise, we shall use observation methods to trace the run.

(a) Use LOG4J to create a trace. Log all invocations of all methods.

(b) Using appropriate logging levels, allow users to follow the execution of:

- Each test (i.e., a test is being carried out)
- Each progress (i.e., the input size has been reduced)
- Start and end of delta debugging only (showing the final result)

(c) Use aspects to create a trace on the console:

- Log all invocations of all methods. Use separate aspects for different methods.
- Extend your example to use LOG4J.

(d) Modify the example in Example 8.2 such that it computes an execution *profile*—that is, it records for each instruction how often it was executed.

8.4 You would like to examine a program run as soon as one of the following holds:

- Function `foo()` is reached
- Variable `a[0]` is equal to 2
- `foo()` is reached and `a[0]` is equal to 2
- `foo()` is reached or `a[0]` is equal to 2
- `foo()` is reached and at least one of `a[0]`, `a[1]`, ..., `a[99]` is equal to 2
- All of `a[0]`, `a[1]`, ..., `a[99]` are equal to 2

Assume that the processor has no special debugging support, except for changing individual machine instructions into interrupt instructions.

(a) Sort these queries according to the execution speed of the examined program. Start with the fastest.

(b) Sketch processor support that can make these queries more efficient.
(c) Sketch possible code instrumentation (e.g., adding new code at compilation time) that makes these queries more efficient.

8.5 When stopping a program, the current *backtrace* is a summary of how your program got where it is. It is a sequence of *frames,* where each frame holds the execution context of a called function. The backtrace starts with the currently executing frame (frame 0), followed by its caller (frame 1), and on up the stack. The following is an example of a backtrace, showing the innermost three frames.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
```

Suppose you have a given backtrace as an array of function names. For instance:

```
backtrace[0] == "m4_traceon"
backtrace[1] == "expand_macro"
backtrace[2] == "expand_token"
```

Your task is to instrument a debugger such that the program being examined stops as soon as a specific backtrace or its superset is reached. In this backtrace, the program should stop as soon as `m4_traceon` is reached while `expand_macro` is active, where `expand_macro` was reached while `expand_token` was active. To instrument the debugger, you can use the following functions:

- `set_breakpoint(`*function*, `ENTER/EXIT|)` sets a breakpoint such that execution stops when entering/exiting the function *function*. It returns a unique number for each created breakpoint.
- `delete_breakpoint(`*bp_nr*`)` deletes the breakpoint number *bp_nr*.
- `continue()` starts or resumes execution until the next breakpoint is reached. It returns the number of the reached breakpoint.
  Example: To make the program stop as soon as "`m4_traceon`" is entered, use the following.

```
m4_bp = set_breakpoint("m4_traceon", ENTER);
do {
    bp = continue();
} while (bp != m4_bp);
delete_breakpoint(m4_bp);
```

Design an algorithm in C-like pseudocode that uses the previous functions to make a program stop at a specific backtrace. Be sure to comment your work.

8.6 Mystery time! When being executed on Mac OS, the `bigbang` program shown in Example 8.3 is reported to hang up after issuing the result (rather than terminating normally).

```
$ bigbang
result is: 2
⟨Interrupting execution⟩
$ _
```

What's wrong with this program?

(a) Use logging functions to log the program state. Use a dedicated method for each class.

(b) Use GDB or another interactive debugger to examine the run.

(c) Use DDD to visualize the relationships between the individual elements.

Document all of your steps in a logbook (see Section 6.5 in Chapter 6).

EXAMPLE 8.3: The `bigbang` program. The `Container` class is defined in Example 8.4. The `Element` class is shown in Example 8.5

```
#include <iostream>
using namespace std;

#include "Element.h"
#include "Container.h"

 bool mode = true;  // (1)

int main (int argc, char *argv[]) {
     Element *a = new Element(1);
     Element *b = new Element(2);
     a->setPeer(b);
     b->setPeer(a);
     a->doSomeStuff();

     Container *c = new Container(10, mode);
     // c->add(b); // (2)
     c->add(a);
     c->add(b);

     cout << "result is: " << c->processElements() << '\n';

     delete c;
     return 0;
}
```

EXAMPLE 8.4: The `Container.h` file for the `bigbang` program

```
#ifndef MY_CONTAINER_H
#define MY_CONTAINER_H

#include "Element.h"

class Container {
private:
     bool deleteElements;
     int size;
     Element **elements;

public:
     Container(int sz, bool del)
       : size(sz), deleteElements(del)
     {
        elements = new Element *[size];
        for (int i = 0; i < size; i++)
           elements[i] = 0;
     }
     int processElements() {
          int sum = 0;
          for (int i = 0; i < size; i++)
               if (elements[i])
                    sum += elements[i]->getData();

          return sum;
     }
     bool add(Element* e) {
        for (int i = 0; i < size; i++)
           if (elements[i] == 0) {
              elements[i] = e;
              return true;
           }

        return false;
     }
     virtual ~Container () {
        if (deleteElements)
          for (int i = 0; i < size; i++)
             delete elements[i];

        delete elements;
     }
};
#endif
```

EXAMPLE 8.5: The `Element.h file` for the `bigbang` program

```
#ifndef _ELEMENT_H
#define _ELEMENT_H

class Element {
     int data;
     Element *peer;

public:
     Element (int d)
       : data(d), peer(0)
     {}

     int getData ()              { return data; }
     void setPeer (Element *p) { peer = p; }
     void resetPeer ()         { peer = 0; }

     void doSomeStuff () {
          if (peer != 0) {
              delete peer;
              resetPeer();
          }
     }

     virtual ~Element () {
        if (peer != 0 && peer->peer == this) {
           peer->resetPeer();
          }
     }
};

#endif
```

"If a program can't rewrite its own code," he asked, "what good is it?"

– Ed Nather
The Story of Mel (1983)