A TUTORIAL INTRODUCTION TO GCT

Brian Marick
Testing Foundations

Documentation for version 1.4 of GCT.
Document version 1.4

This document is a step-by-step guide to the basics of using the Generic Coverage Tool (GCT). The manual pages and the *Generic Coverage Tool (GCT) User's Guide* have the full details.

Please suggest improvements to this manual, to other manuals, or to GCT itself.

Preface

For more information about GCT, other products, or other services, contact:

Brian Marick
Testing Foundations
809 Balboa
Champaign, Illinois 61820

(217) 351-7228
Email: marick@cs.uiuc.edu, testing!marick@uunet.uu.net

You can join the GCT mailing list by sending mail to gct-request@cs.uiuc.edu.

CHAPTER 1

Introduction

In this tutorial, you'll step through the basics of using GCT:

(1) Instrumenting a program.

(2) Obtaining coverage data.

(3) Writing new tests to increase coverage.

(4) Changing the program and reinstrumenting.

Most of the ideas will be introduced as they're used, but a summary of what GCT does seems in order.

A perfectly effective test suite would find every bug. Since we don't know how many bugs there are, we can't measure how closely a test suite approaches perfection. Consequently, we use some approximate measure of test suite quality: since we can't measure what we want, we measure something related.

With coverage, we estimate test suite quality by examining how thoroughly the tests exercise the code:

(1) Is every `if` statement taken in both the true and false directions? (If it's never taken in the true direction, why do you believe that case works?)

(2) Is every `case` taken? What about the `default` case?

(3) Is every `while` loop executed more than once? Does some test force the `while` loop to be skipped? (That is, force the loop test to be false the first time it's tested.)

(4) Is every loop executed exactly once? (There are potential initialization bugs that can be masked by later iterations of the loop.)

(5) Do the tests probe off-by-one errors?

GCT answers these questions; equivalently, it measures branch, loop, and relational operator coverage. These types of coverage are most appropriate near the time the code is written ("unit testing"). Other types of coverage are better suited for later testing ("system testing"), but they aren't described here. The principles are the same.

One of these principles is that coverage numbers must be used carefully. Approximate measures of quality are dangerous; it's easy to concentrate on the "making the numbers", and somehow losing quality in the process. By analogy, consider hiring a college graduate. A perfect grade point average from the best school in the country means something quite different than one from <insert your alma mater's traditional rival here>. You wouldn't hire based only on GPA; don't judge test quality only on coverage.

In particular, it is usually unwise to generate tests solely to achieve high coverage. While it is true that good tests imply coverage, the converse is not necessarily true. Many of the important bugs in programs are caused by omitted code (such as missing error handling) -- designing tests only to exercise code may do a poor job of finding those bugs.[1]

Because the focus of this tutorial is the mechanics of GCT, little more will be said on this very important topic.

Here are some references about the prevalence of different types of errors. Robert L. Glass, ''Persistent Software Errors'', *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981. V. Basili and D. Weiss, ''A Methodology for collecting valid software engineering data'', *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 728-738, November, 1984. Thomas J. Ostrand and Elaine J. Weyuker, ''Collecting and Categorizing Software Error Data in an Industrial Environment'', *Journal of Systems and Software*, Vol. 4, 1984, pp. 289-300. Dewayne E. Perry and W. Michael Evangelist, ''An Empirical Study of Software Interface Errors'', *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, pages 32-38.

CHAPTER 2

Instrumenting Your Program

1. Using lc

The first step is to copy the source from the GCT *demo* directory and its subdirectories to your own directory. (The person who installed GCT can tell you where the demo files are.)

Take a look at the directory. You'll see many files. Some of the more important are:

lc.h, lc.c, and get.c
: This is the source for the program you'll instrument. It's a program that counts the lines of code and comments in C programs.

lc.1
: This is the manpage, suitable for formatting with **nroff**. *lc.1t* is more useful when using **troff** output for printing.

Makefile
: This is the makefile for the program.

run-suite
: This is a test suite driver for **lc**. The test data are in the files named *lc1*, *lc2*, and so on. This test suite was developed using techniques described elsewhere[1], but a few tests were removed for this tutorial.

Type

```
% make
% lc lc.c
```

to see how **lc** works. You should see this output:

```
% lc lc.c
         Pure     Pure       Both              Total    Total      Total
         Code     Comment    Cod&Com  Blank    Code     Comment    Lines    Pages

lc.c     164      305        19       61       183      324        549      19
```

"Pure Code" means lines of code with no comments, "Pure Comment" means lines with only comments; "Both Cod&Com" means lines containing both. "Blank" lines contain neither code nor comments.

2. Deciding What to Measure

The first step is to tell GCT what to measure. That information is kept in the file *gct-ctrl*, which contains:

```
(coverage branch multi loop relational)
(options instrument)
```

The first line tells GCT we want branch, multi-condition, loop, and relational operator coverage. These are the types of coverage I recommend for detailed testing (unit and subsystem testing). We'll see what these coverages mean when we look at GCT's reports. Other types of coverage are possible; see the User's

[1] Brian Marick, ''Experience with the Cost of Test Suite Coverage Measures'', *Pacific Northwest Software Quality Conference*, October, 1991. Also available as a compressed postscript file in cs.uiuc.edu:/pub/testing/experience.ps.Z.

Guide.

The second line tells GCT to measure coverage for all files. We could also name *lc.c* and *get.c* specifically.

3. Instrumenting the Program

The next step is to *instrument* the program so that it collects data about its execution as it runs. We will do this by substituting GCT for the normal C compiler when running the makefile.

3.1. Preparation

Before doing that, we need to make sure that the Makefile will recompile both files in the program:

```
% make clean
```

Since this is the first time we're instrumenting these files, we need to initialize GCT:

```
% gct-init
```
[2]

This program copies a number of files into your directory. Most users will only care about two GCT files:

GCTLOG
: The logfile contains the raw coverage information from one or more tests. The name can be changed; see the User's Guide. There's no logfile yet, because we haven't run any tests.

gct-map
: The mapfile contains the information used to translate the logfile into comprehensible text. Its name can also be changed.

You may be curious about what the other files are:

gct-ps-defs.h
: A makefile usually invokes the C compiler separately for each source file. GCT is invoked in the same way. This file stores information about where the previous invocation left off.

gct-ps-defs.c
: For speed, the coverage data is stored in the memory of the instrumented program. This file defines that in-core log; it is linked into the final executable.

gct-write.c
: To be useful, the in-core log must be written into the logfile. This file contains a routine that does that. It also contains a routine that reads in the logfile, so that the log can accumulate over several tests.

gct-defs.h
: This file contains various C macros used to manipulate the in-core log. It is included by all instrumented files.

---

[2] If the shell cannot find **gct-init**, the directory that GCT was installed in must not be in your search path ($PATH). Ask the person who installed GCT where it is.

3.2. Instrumentation

In this step, we tell the makefile to compile the program in the normal way, but to use GCT as the "compiler". This is done by calling the makefile like this:

```
% make CC=gct
```

You should see something like this:

```
gct   -c  lc.c
gct   -c  get.c
gct -o lc lc.o get.o
```

make gives *lc.c* to GCT. The result is an instrumented object file, *lc.o*. Although GCT appears to be a C compiler, what it is actually doing is instrumenting *lc.c*, placing the result into a temporary file, calling the system's C compiler to compile that file, and leaving the result in *lc.o*.

Several options provide finer control over what GCT does with the instrumented source, which C compiler it uses, and so on. See the User's Guide.

CHAPTER 3

A First Pass Over the Coverage Data

To run the test suite, type

```
% run-suite
```

It will print lines like this to the screen:

```
== T1
== T2
```

Any line not beginning with an equal sign signals a test failure. (There shouldn't be any.)

Now that tests have run, the logfile, *GCTLOG*, exists. To get a first idea of how thorough the test suite is, type

```
% gsummary GCTLOG
```

You should see this:

```
testing-182% gsummary GCTLOG
BINARY BRANCH INSTRUMENTATION (70 conditions total)
2 ( 2.86%) not satisfied.
68 (97.14%) fully satisfied.

SWITCH INSTRUMENTATION (14 conditions total)
0 ( 0.00%) not satisfied.
14 (100.00%) fully satisfied.

LOOP INSTRUMENTATION (18 conditions total)
1 ( 5.56%) not satisfied.
17 (94.44%) fully satisfied.

MULTIPLE CONDITION INSTRUMENTATION (42 conditions total)
2 ( 4.76%) not satisfied.
40 (95.24%) fully satisfied.

OPERATOR INSTRUMENTATION (21 conditions total)
1 ( 4.76%) not satisfied.
20 (95.24%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (165 total)
6 ( 3.64%) not satisfied.
159 (96.36%) fully satisfied.
```

I'll explain the first two sections; the other types of instrumentation are covered in the next chapter.

Every branching statement (if, ?, or loop test) generates two *coverage conditions*: one that the true case be taken, and one that the false case be taken. In **lc**, there are 35 branching statements generating 70 conditions. Of these, two have not been satisfied; two cases (true or false) remain to be exercised.

lc also contains some switch statements with a total of 14 cases (including defaults). All the cases have been taken. (Each case label is a separate coverage condition.)

The summary coverage is high. That's not surprising, since this test suite was derived by removing a few tests from a test suite that reached 100% coverage. However, a summary coverage greater than 90% is typical of thorough test suites.[1]

Simple numbers like **gsummary's** are a blunt instrument. Don't use GCT to reduce your testing to a few numbers, but rather to expand your understanding of what your test suite really does. **greport**, described in the next chapter, is the tool you'll use.

[1] In addition to the *Experience* paper mentioned earlier, I can send an unpublished case study: the coverage of GCT's own test suite.

CHAPTER 4

Writing New Tests

1. Analysing missed coverage

Type

```
% greport GCTLOG
```

You should see:

```
"lc.c", line 137: operator > might be >=. (L==R)
"lc.c", line 137: condition 1 (argc, 1) was taken TRUE 43, FALSE 0 times.
"lc.c", line 139: condition 1 (<...>[...], 1) was taken TRUE 0, FALSE 14 times.
"lc.c", line 153: if was taken TRUE 0, FALSE 29 times.
"lc.c", line 162: loop zero times: 0, one time: 19, many times: 10.
"lc.c", line 172: if was taken TRUE 0, FALSE 43 times.
```

You'll notice that this looks a lot like error output from a C compiler. This is intentional, for two reasons:

(1) Psychological. As with a compiler's output, you examine the errors (in this case, test deficiencies), fix them, redo the compile (rerun the test suite), and repeat the process until you get no more output. Just as with a compiler, the messages you see may prompt large changes, larger than is strictly necessary to make them go away.

(2) Practical. These error messages are compatible with the **error(1)** program (on BSD UNIX) and the GNU Emacs `next-error` function. (A version of `next-error` tailored to GCT is provided with GCT. If you're an Emacs user, you may want to look at Appendix A now.)

Let's examine the output line by line. The first two lines of **greport** output refer to line 137, which is the option processing loop. The line in question is highlighted below, but I recommend you edit *lc.c*; the discussion will be easier to follow if you don't have to constantly flip pages of this tutorial.

```
while (--argc > 0 && (**++argv == '-'))
{
        if (  (*argv)[1] == LCURL || (*argv)[1] == RCURL)
        {
           white_bracket = TRUE;
        }
        else if (sscanf (*argv + 1, "%d", &page_size) == FALSE)
        {
           fprintf (stderr, "lc: Bad page size argument: %s\n", *argv);
           exit (BAD_FLAG);
        }
}
```

The first **greport** line,

```
"lc.c", line 137: operator > might be >=. (L==R)
```

is from *relational* coverage, which checks for tests that probe common misuses of relational operators. In this case, we're worried that the programmer made the mistake of using `>` instead of `>=`. The parenthetical remark (L==R) is suggesting that the unexercised boundary condition, `--argc==0`, is the best way to find this hypothetical bug.

Why? Suppose that > is in fact wrong. We want to write the test with the best chance of causing this program to fail. If we choose --argc==0, the program we have will not enter the while loop, whereas the correct program (with >=) would. For all other possible values, the given program would take the same path as the correct program, so it is less likely to fail.

To force this case, the call to **lc** must have no non-option arguments. It must look like one of these:

```
% lc < INPUT
% lc -} < INPUT
% lc -34 < INPUT
```

If you look at the **run-suite** file, you'll see that there are no such tests: **lc** is never tested when it takes its input from standard input. This is a major omission. Rather than writing a test like this right away, we should write down "input from standard input" as a *test condition* in a separate list of test conditions. We may later be able to combine several of these test conditions into a single test case, which saves us effort.[1]

A relational operator can produce up to three lines of **greport** output. See the *Generic Coverage Tool (GCT) User's Guide* for descriptions of the other two.

The next line of **greport** output is from the same line of the program.

"lc.c", line 137: condition 1 (argc, 1) was taken TRUE 43, FALSE 0 times.

This is from *multicondition* coverage, which checks that all parts of a logical expression are used. Line 137's while loop test has two components:

```
--argc > 0
```

and

```
**++argv == '-'
```

The first of them was always true in every test, never false. The second component isn't mentioned, because it's evaluated to both true and false. If you want to see how many times it's evaluated to each value, use **greport -all** (see the manpage).

This condition tells us the same thing that we already knew. To get --argc equal to zero, there can be only arguments beginning with a dash -- that is, **lc** must take its input from standard input.

---

[1] It's also likely to result in a better test -- more complicated test cases are better (up to a point) because they are more likely to catch bugs by chance.

The next line,

"lc.c", line 139: condition 1 (<...>[...], 1) was taken TRUE 0, FALSE 14 times.

corresponds to this code:

```
while (--argc > 0 && (**++argv == '-'))
{
        if (  (*argv)[1] == LCURL || (*argv)[1] == RCURL)
        {
           white_bracket = TRUE;
        }
        else if (sscanf (*argv + 1, "%d", &page_size) == FALSE)
        {
           fprintf (stderr, "lc: Bad page size argument: %s\n", *argv);
           exit (BAD_FLAG);
        }
}
```

It is also an example of multicondition coverage, but the message looks peculiar. What does (<...>[...], 1) mean? It's there in case "condition 1" is not enough to locate the condition **greport** refers to. (In this case, the condition is (*argv)[1] == LCURL.)

Conditions are numbered from left to right. But in a deeply nested expression, like (A && (B || C) || D), it can be hard to count conditions accurately. The parenthetical remark helps you find them. The first component is the leftmost operand of the condition. In this case, it's (*argv[1]). **Greport** always abbreviates arrays as *arrayname*[...], because the expression in brackets is often complex and would make the line too long. When *arrayname* is more complicated than a simple identifier, GCT abbreviates it as <...>, again to save space.

These abbreviation rules are perhaps a bad idea, but in practice it's rarely difficult to figure out what subexpression is meant. As an additional help, the number in parentheses is the nesting depth of the subexpression, starting with 1. (It's always deeply parenthesized expressions that cause trouble.)

What this line tells us is that *argv[1] is never LCURL. (LCURL is defined in *lc.h*.) That is, we've never given the program the -{ option. A quick check of **run-suite** confirms this. We'll write that down in our test condition list.

The next line,

"lc.c", line 153: if was taken TRUE 0, FALSE 29 times.

tells us that this if has never been taken in the true direction.

```
if (argc == 0)
{
   tally_file (stdin, &file_tally);
   show_header ();
   show_tally ("", &file_tally);
}
```

We already knew that: the program never reads from standard input.

The next line,

"lc.c", line 162: loop zero times: 0, one time: 19, many times: 10.

is the first example of *loop* coverage. This particular loop traverses all of **lc**'s non-option arguments.

```
for (index = 1; index <= argc; index++)
{
   if ((fp = fopen (*argv, "r")) == NULL)
   {
           status = FILE_NOT_FOUND;
           fprintf (stderr, "lc: can't open %s\n", *argv);
   }
   else
   {
           tally_file (fp, &file_tally);
           if (fclose (fp) == EOF)
              panic (PANIC, "Fclose error.");
           show_tally (*argv, &file_tally);
           if (argc > 1)
              make_total (&total_tally, &file_tally);
   }
   argv++;
}
```

A **greport** line for a loop tells about three types of traversals:

(1) ones where the loop test failed on the first try, so the loop body was never entered.

(2) ones where the loop test failed on the second try, so the loop body was entered exactly once.

(3) ones where the loop body was traversed more than once.

Some types of bugs are only detected by one of these cases.

In this particular case, we know that argc>0 (because we just looked at the if statement that handled the zero case). So it is impossible to traverse the while loop 0 times. This is an example of an *infeasible test condition*. We can ignore it.

Of the two feasible cases, we see that **lc** was given a single argument 19 times and more than one argument 10 times.

We move on to:

"lc.c", line 172: if was taken TRUE 0, FALSE 43 times.

which corresponds to this line:

```
if (fclose (fp) == EOF)
   panic (PANIC, "Fclose error.");
```

Evidently, the programmer thinks an EOF return from fclose() is impossible (but is checking anyway, just in case). Of course, what a programmer thinks and what is true may be two different things. We as testers would want to think hard about how to generate EOF returns. (Error handling is a fertile source of serious bugs caused by mistaken assumptions.) Let's assume that we'll fail, that this is another example of an infeasible test condition.

We're finished. We generated only a short list of things to test. This is as it should be, given a thorough starting test suite. We wasted some time looking at impossible conditions, but not very much.

2. The new tests

We have two new test conditions:

1. input from standard input.
2. Use the -{ option.

This is easy to do with a single test, which you should just type to the shell:

```
% echo "{" | lc "-{"
```

(Notice that braces are quoted to prevent possible interpretation by your shell.) This is not a very thorough test of the -{ option, but it will do for this tutorial. Is the output correct?

Now type

```
% gsummary GCTLOG
```

You'll see that total coverage has gone up to 98.79%. greport GCTLOG shows:

"lc.c", line 162: loop zero times: 0, one time: 19, many times: 10.
"lc.c", line 172: if was taken TRUE 0, FALSE 43 times.

We know that both of these are infeasible.

3. Suppressing infeasible coverage

It's annoying that we can't make the infeasible test conditions go away. Worse, if we're testing a large program - one where it may take us many tries to eliminate all feasible test conditions - we don't want to waste time looking at infeasible conditions every time we run **greport** after a change to the test suite.

The **gedit** program can help. To use it, type

```
% greport -edit GCTLOG > edit.g
```

Edit *edit.g*. You'll see this:

"lc.c", line 162: [25: 0 19 10] loop zero times: 0, one time: 19, many times: 10.
"lc.c", line 172: [34: 0 43] if was taken TRUE 0, FALSE 43 times.

The numbers in brackets are the raw versions of what the text says. In the first case (line 162), the 0 is the number of times the loop was skipped. The next two numbers are the numbers the loop was taken once and many times. In the case of branches, like line 172, the first number is the true count, the second the false count.

Replace the zeros that are impossible with either "s", "S", "0s", or "0S":[2]

"lc.c", line 162: [25: 0S 19 10] loop zero times: 0, one time: 19, many times: 10.
"lc.c", line 172: [34: s 43] if was taken TRUE 0, FALSE 43 times.

[2] If you're using Emacs "gedit-mode", SPC will helpfully position you at the first zero in the next line.

Now type

```
% gedit edit.g
% gsummary GCTLOG
```

You’ll see:

BINARY BRANCH INSTRUMENTATION (70 conditions total)
0 (0.00%) not satisfied.
70 (100.00%) fully satisfied. **[1 (1.43%) suppressed]**

SWITCH INSTRUMENTATION (14 conditions total)
0 (0.00%) not satisfied.
14 (100.00%) fully satisfied.

LOOP INSTRUMENTATION (18 conditions total)
0 (0.00%) not satisfied.
18 (100.00%) fully satisfied. **[1 (5.56%) suppressed]**

MULTIPLE CONDITION INSTRUMENTATION (42 conditions total)
0 (0.00%) not satisfied.
42 (100.00%) fully satisfied.

OPERATOR INSTRUMENTATION (21 conditions total)
0 (0.00%) not satisfied.
21 (100.00%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (165 total)
0 (0.00%) not satisfied.
165 (100.00%) fully satisfied. **[2 (1.21%) suppressed]**

If you use **greport**, it will show you nothing. You’re done using coverage on your test suite.

gedit works by adding information to the mapfile. The user’s manual describes other ways to edit the mapfile to control what information the reporting tools display.

CHAPTER 5

Updating Mapfiles and Logfiles

The facilities described in this chapter are available only with GCT Expansion Kit 1, which is purchased separately. The Expansion Kit provides two facilities:

(1) You can change a single file and incrementally update the mapfile. Without the expansion kit, you must reinstrument and recompile all the files when one of them changes.

(2) As you test a program, you accumulate information: which coverage conditions have been satisfied, and which you've ruled out as infeasible. This information is stored in the logfile and by **gedit**'s changes to the mapfile. In theory, a new version of the program that differs in only a single character could invalidate all that information. Without the expansion kit, old information must be discarded when each new version is produced. With it, you can assume that the old information is still (mostly) valid, and update it to match the new version. This allows you to defer worrying about invalidated information until the program's final version. This is more efficient.

The full capabilities of the Expansion Kit are described in the *User's Guide*.

1. Updating the Mapfile

Here, we'll see how the mapfile is updated after a single file changes. Edit main() in *lc.c*. Add the line

```
argc = argc + 1 - 1;
```

anywhere in the routine. (We want to add a line that does nothing, so the tests don't fail.)

Recall that main is the routine where we suppressed two infeasible coverage conditions: that fclose can fail and that the file-processing for loop be entered zero times.

Once again, type

```
% make CC=gct
```

You should see something like this:

```
gct   -c  lc.c
gct -o lc lc.o get.o
```

Notice that get.c was not reinstrumented because it hasn't been changed.

Now type

```
% gsummary GCTLOG
```

You'll see something like this:

```
The mapfile and logfile come from two different instrumentations.
The mapfile comes from one begun on Sun Aug 23 09:56:15 1992.
The logfile comes from one begun on Sun Aug 23 09:48:24 CDT 1992.
```

Data from an old logfile might be invalid for this changed source, so **gsummary** complains. The next section will describe how to update the logfile. For now, create a new log file by rerunning the old test suite and the new test we designed.

```
% rm GCTLOG[1]
% run-suite
% echo "{" | lc "-{"
% gsummary GCTLOG
```

gsummary should show you this:

BINARY BRANCH INSTRUMENTATION (70 conditions total)
0 (0.00%) not satisfied.
70 (100.00%) fully satisfied. [1 (1.43%) suppressed]

SWITCH INSTRUMENTATION (14 conditions total)
0 (0.00%) not satisfied.
14 (100.00%) fully satisfied.

LOOP INSTRUMENTATION (18 conditions total)
0 (0.00%) not satisfied.
18 (100.00%) fully satisfied. [1 (5.56%) suppressed]

MULTIPLE CONDITION INSTRUMENTATION (42 conditions total)
0 (0.00%) not satisfied.
42 (100.00%) fully satisfied.

OPERATOR INSTRUMENTATION (21 conditions total)
0 (0.00%) not satisfied.
21 (100.00%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (165 total) 0 (0.00%) not satisfied.
165 (100.00%) fully satisfied. [2 (1.21%) suppressed]

Even though main has changed, the two suppressed conditions remain suppressed. We are assuming that they remain impossible. This is a reasonable assumption to make during code and test development. You want to reexamine the impossibility of those conditions after all changes are made, instead of as each change is made.

GCT retained suppression because the change was a *minor* change, defined as one that didn't change the instrumentation of the routine. Had the change been major - adding an if statement, for example - GCT would have forced you to reevaluate the two impossible conditions. They would not have been suppressed in the updated mapfile, so they would have reappeared in **greport** output.

You can instruct GCT to force reevaluation after minor changes, or even whenever any change is made to any routine. See the *User's Guide*.

2. Updating the Logfile

In addition to updating the mapfile, you may also want to update the logfile. As an example of this, edit *get.c*. In the routine skip_token, you'll see a commented-out if statement. Remove the comments to produce a changed routine. Notice that this is a major change.

---

[1] **run-suite** actually deletes GCTLOG so that the log contains information only about that run. But an explicit removal is clearer for this tutorial.

Reinstrument and update the mapfile, first saving a copy of the old information:

```
% cp gct-map gct-map.save
% cp GCTLOG GCTLOG.save
% make CC=gct
```

Now update the log:

```
% gct-newlog gct-map.save GCTLOG.save > GCTLOG
```

You should see

```
Routine skip_token (in get.c) has changed too much; its entries will be zeroed.
```

All information about how the test suite exercises skip_token has been lost, because it is very likely meaningless for the new routine.[2] However, the information for the rest of the program remains. Type

```
% greport GCTLOG
```

and you'll see that only the lines from the changed routine appear:

"get.c", line 148: if was taken TRUE 0, FALSE 0 times.
"get.c", line 150: while was taken TRUE 0, FALSE 0 times.
"get.c", line 150: loop zero times: 0, one time: 0, many times: 0.
"get.c", line 151: condition 1 (ch, 4) was taken TRUE 0, FALSE 0 times.
"get.c", line 151: condition 2 (ch, 4) was taken TRUE 0, FALSE 0 times.
"get.c", line 152: condition 1 (ch, 3) was taken TRUE 0, FALSE 0 times.
"get.c", line 152: condition 2 (ch, 3) was taken TRUE 0, FALSE 0 times.
"get.c", line 153: condition 1 (ch, 2) was taken TRUE 0, FALSE 0 times.
"get.c", line 153: condition 2 (ch, 2) was taken TRUE 0, FALSE 0 times.
"get.c", line 154: condition 1 (ch, 1) was taken TRUE 0, FALSE 0 times.
"get.c", line 154: condition 2 (ch, 1) was taken TRUE 0, FALSE 0 times.

Had any of the lines in the routine been suppressed, that suppression would have been forgotten as well.

Since we changed this routine, we're presumably about to rerun all its tests again, so losing its coverage information is no hindrance. However, if we were also in the midst of testing another unrelated routine, we'd be glad that that routine's coverage information was retained.

Once the program was finished and tested, we'd typically reinstrument completely, rerun all the tests, and check coverage one last time.

3. Utility scripts

Because information about your testing of a program is distributed among several files (the current executable, the source it was built from, the mapfile and its edits, and one or more logfiles), it's unfortunately easy for those files to become inconsistent. Unless you make fewer stupid mistakes than I do, using simple shell scripts to keep checkpoints of GCT files will save you pain. This section describes some recommended shell scripts that you can adapt to your needs.

---

[2] In this particular case, the old coverage is actually still valid, since the if doesn't affect the routine.

3.1. Basic use

First, copy the scripts into the demo directory:

```
% cp kit1/* .
```

To instrument the entire program for the first time, use **instrument**.

```
% instrument
```

You'll see

```
Save current instrumented state of program?
```

Answer 'n' (without the quotes) because there isn't a current state the first time you instrument. You should then see something like

```
gct -c lc.c
gct -c get.c
gct -o lc lc.o get.o
FINISHED: Use 'update' if you want to update the logfile.
```

Run the test suite (**run-suite**), and use **gedit** to edit the mapfile. Then change *lc.c*, but make a syntax error.

To reinstrument the changed file, type

```
% reinstrument
```

Now that you have a logfile and edited mapfile, you should answer 'y'. You'll get a compile error:

```
lc.c: In function main:
lc.c:158: 'x' undeclared (first use this function)
lc.c:158: (Each undeclared identifier is reported only once
lc.c:158: for each function it appears in.)
lc.c:158: parse error before 'show_tally'
*** Error code 1
make: Fatal error: Command failed for target 'lc.o'
```

Fix the error and reinstrument again. This time, answer 'n' when asked if you want to save the state. For one thing, you just saved it. For another, the mapfile has been changed by the incomplete instrumentation, but the logfile has not been updated, so the two files are no longer consistent.

Next, type

```
% update
```

This produces a GCTLOG that matches the reinstrumented program. You can now continue your testing.

3.2. Recovery

Now let's consider a case where you accidentally destroy some data. Type

```
% gclean
```

This wipes out all GCT files, including the mapfile. You now have a logfile with no way of interpreting it. If you didn't know that, you might try reinstrumenting:

```
% reinstrument
Save current instrumented state of system?
```

```
y
cp: gct-map: No such file or directory
```

Investigating this error, you discover what's happened. **reinstrument** normally makes the checkpoint in *save*, first saving the previous checkpoint in *save2*. Because the current state is inconsistent, the checkpoint directory is empty. What you need to do is make a consistent state in the build directory, then update it with the information in *save2*. Type this:

```
% instrument          # answer 'n'
% cp save2/* save     # Recover works on this directory
% recover
```

You should now have a logfile and mapfile corresponding to the latest version of your program, including all applicable edits and coverage information. Unless you keep good records, the coverage information might not be that useful, since you probably won't know what tests it corresponds to. But not having to rethink and reapply the edits will be a relief.

APPENDIX A

Using gedit.el

gedit.el provides a version of the Emacs next-error interface that works better for **greport** output. The source for *gedit.el* is, by default, in the GCT library directory; whoever installed GCT may have put it with other local emacs files.

The most convenient way to use *gedit.el* is to always put **greport** output into files ending in .g and put something like

```
(setq auto-mode-alist (cons (cons "\\.g$" 'gedit-mode) auto-mode-alist))
(autoload 'gedit-mode (expand-file-name "~/gct/src/gedit.el"))
```

into your *.emacs* file. That done, whenever you edit such a file, you'll use gedit-mode.

When in gedit-mode, each time you hit SPC, a new **greport** line will move to the top of one window. The appropriate source file will appear in the other window. An arrow (=>) will point to the line in question. (The arrow is not part of the file.)

Like next-error, there's no convenient way to back up. If you hit e, you'll restart from the beginning.