The GNU Coverage Tool
A Brief Tutorial[*]

Peter H. Fröhlich
phf@cs.ucr.edu

Revision 1.8
September 13, 2003

Abstract

The goal of this tutorial is to introduce you to `gcov`, the **G**NU **cov**erage tool, part of `gcc`, the **G**NU **c**ompiler **c**ollection. Completing the tutorial should take at *most* one hour, after which you will be able to use `gcov` on your own to measure *test coverage*. However, there are many more uses for `gcov` than we can go over here. Please refer to the official documentation [1] if you are interested in learning more.

1 Introduction

This tutorial is part of an archive that contains a simple C++ program `xmpl.cpp` as well as a `Makefile` to build that program for use with `gcov`. You'll begin by reviewing these two files (and I mean "reviewing" by opening them in some editor) and building the executable.

1.1 Take a look at `xmpl.cpp`

The C++ program `xmpl.cpp` consists of two major parts (see Figure 1). The first part converts the basic C-style command-line arguments (which are passed as parameters to `main`, usually called `argc` and `argv`) into more convenient C++ abstractions (an `std::vector` of `std::string` objects).[1] To keep things simple there is a constant bound for the number of command line arguments that can be handled. This is *not* a good idea for production quality software—which should of course be able to deal with *any* number of arguments—but at least there is an assertion that will result in an error message if too many arguments are passed.

The second part of the program iterates over the arguments and tries to establish "pair-wise equality" or some such notion. The details are not really relevant, but you can see that it checks a pair of arguments first, printing a message if they are equal. If they are not equal, it checks for the string "Hey!" instead and prints a differ-

[*] I am interested in improving this tutorial, so please email me any comments or questions you might have.

1. There is no "special" reason for this, it just makes the second part simpler to write: We don't have to use C functions like `strcmp` that are easy to get wrong; instead we can use the operator `==` for `std::string` objects.

```
// $Id: xmpl.cpp,v 1.1 2003/09/06 21:57:18 phf Exp $

#include <string>
#include <iostream>
#include <vector>
#include <cassert>

const int MAXARGS = 16;

int main( int argc, char* argv[] ) {

  // Convert C-style arguments to C++ abstractions.
  assert( argc <= MAXARGS );
  std::vector<std::string> args( MAXARGS );
  for (int i = 0; i < argc; i++) {
    std::string s( argv[i] );
    args[i] = s;
  }

  // Establish pair-wise equality between arguments.
  for (int i = 1; i < argc-1; i++) {
    if (args[i] == args[i+1]) {
      std::cout << "Check!" << std::endl;
    }
    else if ( args[i] == "Hey!" ) {
      std::cout << "Huh?" << std::endl;
    }
    else {
      std::cout << "Oops!" << std::endl;
    }
  }

  return 0;
}
```

Figure 1: The example program `xmpl.cpp` for the `gcov` tutorial.

ent message; if there is neither pair-wise equality nor the string “Hey!” it prints yet another message.[2] This completes our tour of the example program.

1.2 Take a look at `Makefile`

The `Makefile` for this tutorial is pretty simple (see Figure 2). The `CPPFLAGS` definition overrides the default options for the C++ compiler `gcc`. Only the options `-fprofile-arcs` and `-ftest-coverage` are important for using `gcov`, the other options deal with warnings and optimizations.

Note that *all* source files for which you want to measure coverage have to be compiled with these options; here we only have one such file, but in general your program will consist of multiple source files. The `-f` options tell the compiler to generate additional code measuring which parts of the program were actually executed during a specific run.

```
# $Id: Makefile,v 1.2 2003/09/13 10:58:38 phf Exp $

CPPFLAGS=-Wall -W -pedantic -O0 -fprofile-arcs -ftest-coverage

xmpl:   xmpl.cpp

.PHONY: clean zip
clean:
        rm -f xmpl *.bb *.bbg *.gcov *.da gmon.out *.tar.gz
zip:    README Makefile xmpl.cpp gcov.pdf
        tar czvf gcov-tutorial.tar.gz $ˆ
```

Figure 2: The `Makefile` for the `gcov` tutorial.

The `xmpl` target says that to build our program, we need to “handle” the file `xmpl.cpp` in some way. We rely on a built-in rule of `make` that maps files ending with `.cpp` to the C++ compiler. The `clean` target deletes some temporary files, while the `zip` target creates the archive for the tutorial.[3]

1.3 Run `make`

Do an `ls` command to make sure that you only have the files `xmpl.cpp` and `Makefile` so far. Now run `make`. After it is done, do an `ls` again.

Aside from the executable `xmpl`, the files `xmpl.bb` and `xmpl.bbg` have been created as well. These files were produced by the `-f` options and contain information about the structure of the *machine code* generated for `xmpl` and the correspondence of “chunks” of machine code to line numbers in the C++ source code.[4]

2. There is no “special” meaning hidden here either, it’s just an *example* program after all.

3. Making these targets *phony* means that they do not generate files according to the usual `make` rules.

4. The `.bb?` extension stands for *basic block*, a chunk of machine code with one entry and one exit only. As far

```
Could not open data file xmpl.da.
Assuming that all execution counts are zero.
  0.00% of 1 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/new
Creating new.gcov.
  0.00% of 1 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/iostream
Creating iostream.gcov.
  0.00% of 3 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_iterator.h
Creating stl_iterator.h.gcov.
  0.00% of 4 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_construct.h
Creating stl_construct.h.gcov.
  0.00% of 11 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_uninitialized.h
Creating stl_uninitialized.h.gcov.
  0.00% of 4 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_alloc.h
Creating stl_alloc.h.gcov.
  0.00% of 2 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/basic_string.h
Creating basic_string.h.gcov.
  0.00% of 14 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_vector.h
Creating stl_vector.h.gcov.
  0.00% of 14 source lines executed in file
  xmpl.cpp
Creating xmpl.cpp.gcov.
```

Figure 3: The initial output of `gcov` (probably different on your system).

2 Measuring Coverage

Now we're ready to start measuring coverage for the example program. Before you continue, make sure that you did *not* run `xmpl` yet; if you *did* run it, do a `make clean` and a `make` to continue with a clean slate.

2.1 Run `gcov` on `xmpl`

Run `gcov` for the first time, simply by using the command `gcov xmpl`.[5] This will produce a lot of messages about various files, including some that look like errors (see Figure 3).

First there's no `xmpl.da` file yet. This file is used to record which branches were taken in the machine code for `xmpl` during a particular execution. Since we did not run `xmpl` yet, the file does not exist. Next there's a list of `0.00% lines executed` messages, mostly for STL sources, but finally also one for the file

```
        // $Id: xmpl.cpp,v 1.1 2003/09/06 21:57:18 phf Exp $

        #include <string>
        #include <iostream>
        #include <vector>
        #include <cassert>

        const int MAXARGS = 16;

######  int main( int argc, char* argv[] ) {

          // Convert C-style arguments to C++ abstractions.
######    assert( argc <= MAXARGS );
######    std::vector<std::string> args( MAXARGS );
######    for (int i = 0; i < argc; i++) {
######      std::string s( argv[i] );
######      args[i] = s;
          }

          // Establish pair-wise equality between arguments.
######    for (int i = 1; i < argc-1; i++) {
######      if (args[i] == args[i+1]) {
######        std::cout << "Check!" << std::endl;
            }
######      else if ( args[i] == "Hey!" ) {
######        std::cout << "Huh?" << std::endl;
            }
            else {
######        std::cout << "Oops!" << std::endl;
            }
          }

######    return 0;
######  }
```

Figure 4: Initial content of `xmpl.cpp.gcov`.

`xmpl.cpp` itself. Again, these numbers are 0 for now since we did not run `xmpl` yet.

If you do an `ls`, you'll see that `gcov` has produced a number of log files for these source files. For example, open `xmpl.cpp.gcov` in an editor of your choice and take a look around (see Figure 4). The log files essentially contain source code annotated with a lot of ###### symbols in several lines. Note how these symbols *only* occur in lines that actually contain code, i.e. things that would "do stuff" if `xmpl` were executed.

as `gcov` is concerned, basic blocks are the places where "stuff gets done" between jumps. To measure coverage, we have to "monitor" which jumps are taken to find out which basic blocks are executed.

5. If your project consists of *multiple* object files that are linked together separately, you need to run `gcov` on *each* individual object file to obtain complete coverage information.

```
  0.00% of 1 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/new
Creating new.gcov.
100.00% of 1 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/iostream
Creating iostream.gcov.
100.00% of 3 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_iterator.h
Creating stl_iterator.h.gcov.
 25.00% of 4 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_construct.h
Creating stl_construct.h.gcov.
 18.18% of 11 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_uninitialized.h
Creating stl_uninitialized.h.gcov.
100.00% of 4 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_alloc.h
Creating stl_alloc.h.gcov.
  0.00% of 2 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/basic_string.h
Creating basic_string.h.gcov.
100.00% of 14 source lines executed in file
  /usr/include/gcc/darwin/3.1/g++-v3/bits/stl_vector.h
Creating stl_vector.h.gcov.
 64.29% of 14 source lines executed in file
  xmpl.cpp
Creating xmpl.cpp.gcov.
```

Figure 5: The next output of `gcov` (probably different on your system).

2.2 Run `xmpl` and `gcov`

Now run `xmpl` without any arguments and do an `ls` again. Note how there now is a `xmpl.da` file. We ran the program once, so now we know which branches were taken in the machine code during that particular execution. Nothing else seems changed.

However, if you run `gcov xmpl` again, you will get quite different results than before (see Figure 5). For each source file, most importantly for `xmpl.cpp`, we now get "real" percentage numbers for coverage, and also new log files.

If you open `xmpl.cpp.gcov` in an editor again, you will now see ###### in fewer lines, and actual numbers in more (see Figure 6). These numbers essentially mean "how often" a given line was executed (the "6" next to the template instantiation is the result of the template being expanded into more complex code, but the expansion being invisible). Lines that were never executed still have the ###### marker.

```
                // $Id: xmpl.cpp,v 1.1 2003/09/06 21:57:18 phf Exp $

                #include <string>
                #include <iostream>
                #include <vector>
                #include <cassert>

                const int MAXARGS = 16;

           1    int main( int argc, char* argv[] ) {

                  // Convert C-style arguments to C++ abstractions.
           1      assert( argc <= MAXARGS );
           6      std::vector<std::string> args( MAXARGS );
           2      for (int i = 0; i < argc; i++) {
           1        std::string s( argv[i] );
           1        args[i] = s;
                  }

                  // Establish pair-wise equality between arguments.
           1      for (int i = 1; i < argc-1; i++) {
      ######        if (args[i] == args[i+1]) {
      ######          std::cout << "Check!" << std::endl;
                    }
      ######        else if ( args[i] == "Hey!" ) {
      ######          std::cout << "Huh?" << std::endl;
                    }
                    else {
      ######          std::cout << "Oops!" << std::endl;
                    }
                  }

           1      return 0;
           1    }
```

Figure 6: Next content of `xmpl.cpp.gcov`.

Run `xmpl` again, still without arguments, then do `gcov xmpl` again and look at the log file. All numbers should have doubled, but the percentages displayed and the ###### markers are still the same. Obviously the file `xmpl.da` is "added to" each time `xmpl` is run. If you want to start your coverage analysis "from scratch" you have to delete the `xmpl.da` file. Try this if you feel like it.

3 Test Coverage

You now know how to use `gcov` to measure coverage in principle. For *testing*, however, our goal is to achieve 100% coverage: We want to make sure that we have enough test cases to execute *every* line of code at least once. Until that's the case, we are *certainly* not done with testing. Even once we have 100% line coverage, we are usually not done, but getting at *least* that is still an important goal.[6] You'll tackle that goal now.

3.1 Starting Out

Do a `make clean` followed by a `make` to get rid of all temporary files and start with a clean slate. Run `xmpl` and `gcov xmpl` again and verify that we still have no coverage inside the second `for` loop (see Figure 6).

3.2 A Little More Coverage

Now let's supply an argument for the first time in hope of getting more coverage. Run `xmpl xxx` followed by `gcov xmpl`. The coverage did not go up. Why?

Try to answer before reading on!

The second `for` loop is only executed if there are at least *two* arguments besides the program name itself. So let's try `xmpl xxx xxx` instead, followed by `gcov xmpl` again. The coverage went up, and in the log file the first `if` inside the `for` has now been executed. Examine `xmpl.cpp.gcov` to verify that.

3.3 More Coverage

Besides two equal arguments, there's also a case for two unequal ones, so let's try `xmpl xxx yyy` next, followed by `gcov xmpl` again. Coverage increased again, and checking the log file we now only have one line left to get 100%, the line dealing with `Hey!` as a parameter. So let's try `xmpl Hey!` next, followed by `gcov xmpl`. Huh? Why does coverage not go up?

Try to answer before reading on!

The reason is that we only check for `Hey!` if at least *two unequal* arguments are given.

Let's verify that with `xmpl Hey!  Hey!` followed by `gcov xmpl`. Again, coverage does not increase, as expected. In other words, we have to use `xmpl xxx Hey!` to get the desired result.

Try that before reading on!

Oops, that did not work either. Looking at the code, we can see that the argument at position `i` is compared to `Hey!`, but in the case of `xmpl xxx Hey!` the word `Hey!` is at position `i+1` instead.

We thus end up with `xmpl Hey!  xxx` as our test case. Now `gcov xmpl` confirms 100% coverage, and the log file does not show any ###### markers anymore.

4 Summary

Following this tutorial, you saw that we need at least three test cases to achieve 100% coverage for `xmpl.cpp`:

```
xmpl xxx xxx
xmpl xxx yyy
xmpl Hey! xxx
```

We also saw that "intuitive" test cases such as `xmpl xxx Hey!` do not always work as we expect at first glance.

6. The reason is that executing each line once is not the same as exercising each *path* through the program once. However, 100% path coverage is—in general—too ambitious: The number of paths through a program grows *exponentially* with the number of decisions it makes. The important thing to remember is that 100% line coverage does *not* mean that your program is bullet proof!

Other test cases are sensible as well, but not for reasons of coverage. Boundary analysis suggests testing

```
xmpl
xmpl xxx
```

and also (since we do white-box testing and thus know about the magic number 16)

```
xmpl xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
  xxx xxx xxx
xmpl xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
  xxx xxx xxx xxx
```

with 15 and 16 arguments respectively. The latter will lead to the assertion failing, but the former should still go through.

Again, note that 100% line coverage does not mean that there are no more sensible test cases, but it is a good basic criterion. As long as your tests do not achieve 100% line coverage, chances are you forgot something. By analysing the `gcov` log files you can get a sense for what other test cases to use, and you can also see if there's "unreachable code" that you still would like to keep in the executable for some reason. Just be sure to note that somewhere.

Acknowledgements

Kudos go to Casey Cobb (CS 100, Spring 2003) for valuable feedback.

References

[1] GNU Project. *gcov: A Test Coverage Program*. Chapter 8 in [2].

[2] GNU Project. *Using and Porting the GNU Compiler Collection (GCC)*. Available at `http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/`.