

A Software Package for Generating Code Coverage Reports With Gcov

Zhenmai Hu

Dr. Michael D. Adams, Supervisor

Dr. Wu-Sheng Lu, Committee Member

Department of Electrical and Computer Engineering

University of Victoria



Structure of This Presentation

1. Introduction

- ❖ The Importance of Software Testing

2. Background

- ❖ Software Testing
 - Software Testing Methods
 - Structural Coverage Analysis
 - Code Coverage Criteria

- ❖ GCC and Gcov

- ❖ Name Mangling

3. The Gcov Report Generator (GRG) Software

- ❖ Software Introduction

- ❖ The GRG Library

- ❖ The Application Program Coverage

- ❖ Usage Example

- ❖ Software Installation

4. Conclusion and Future Work

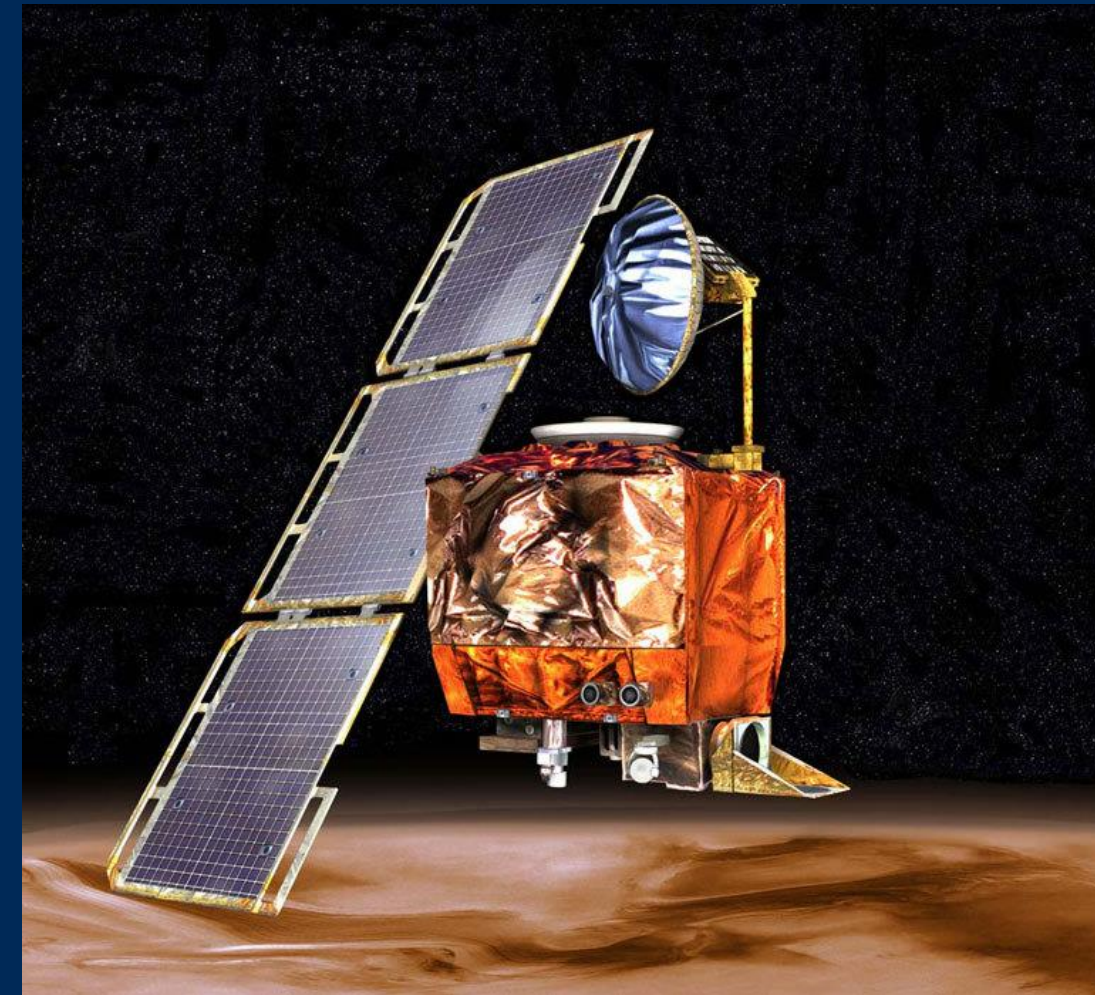
Introduction

❖ The Importance of Software Testing

1. The European Space Agency's Ariane 5 Flight 501 failed forty seconds after takeoff on June 4, 1996.
2. The NASA Mars Climate Orbiter approached Mars at the wrong angle when entering the upper atmosphere and disintegrated in 1998.



Ariane 5 Flight 501 from BBC News by Jonathan Amos



The artist's concept of NASA Mars Climate Orbiter.
Credit: NASA/JPL-Caltech

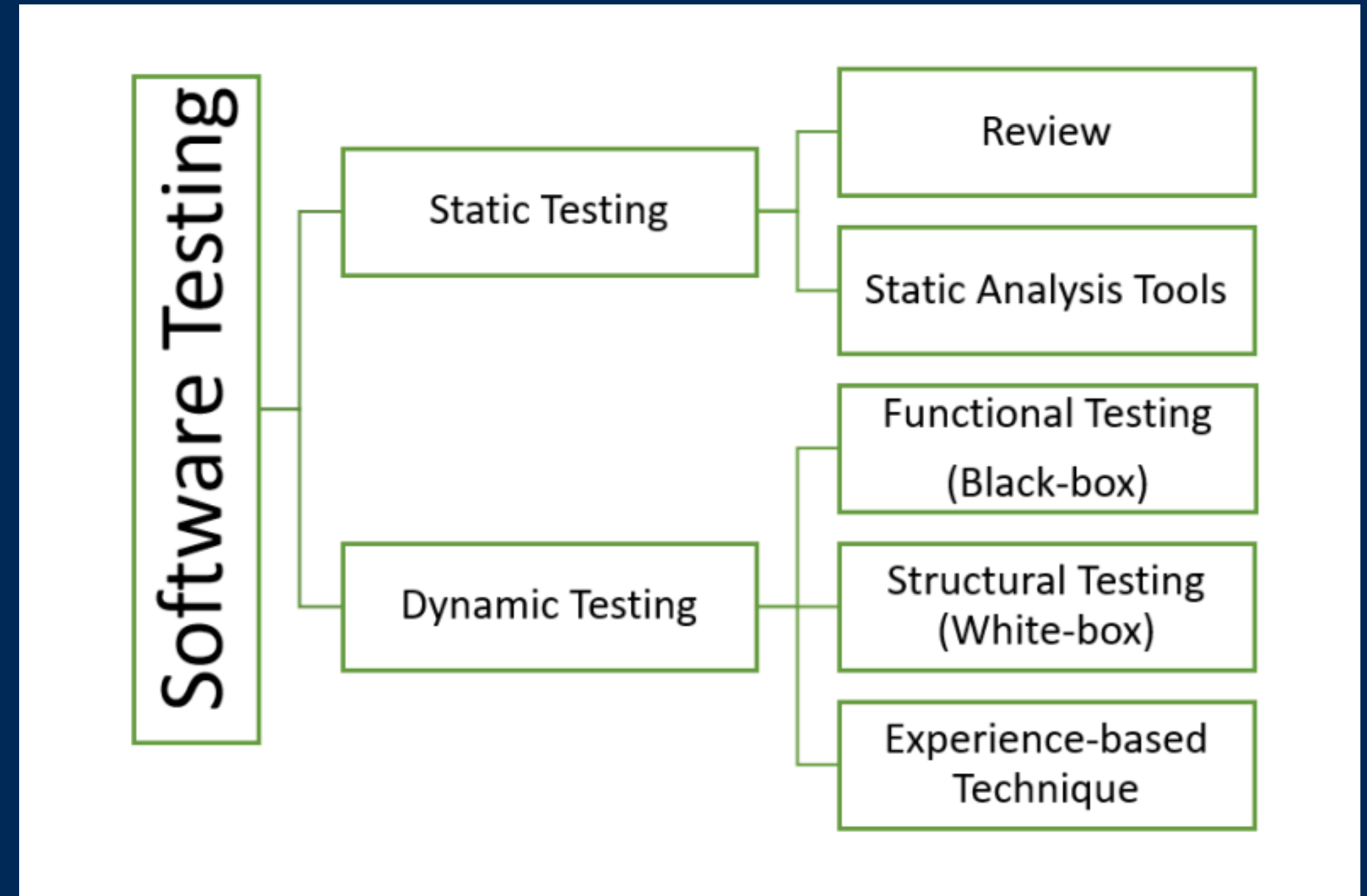
Background - Software Testing

❖ Software Testing Methods

Software testing considers many aspects of software behavior.

For example, testing considers if the software being tested:

- responds correctly to all required inputs;
- has acceptable performance consumption towards time and memory;
- is sufficiently usable for applications;
- works appropriately in all intended environments;
- meets the general needs of users; and
- achieves the design and development requirements without side effects.



Summarization of software testing strategies

Background - Software Testing

❖ Structural Coverage Analysis

Structural coverage analysis is frequently utilized to evaluate testing thoroughness by determining the exercised code structure in testing procedures. It can typically be classified into two categories: **control flow criteria**(the flow of operations and paths performed during software execution) and **data flow criteria**(flow of data through variable assignments and references).

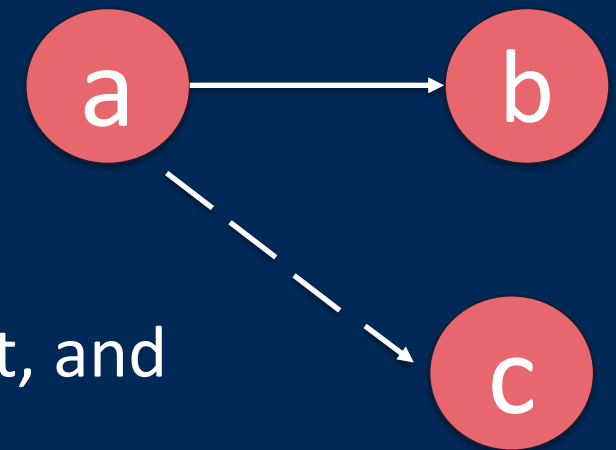
A **control-flow graph (CFG)** is a directed graph depicting all execution paths in code.

Each **node** in a CFG is a straight-line code sequence with one entry point and one exit point, and represents a **basic block** in computer programming.

Each **edge** that connects nodes in a CFG, corresponds to a **branch** in code.

A **condition** is a boolean expression that does not contain boolean operators like AND, OR and NOT.

A **decision** is a boolean expression composed of conditions and zero or more boolean operators.



Background - Software Testing

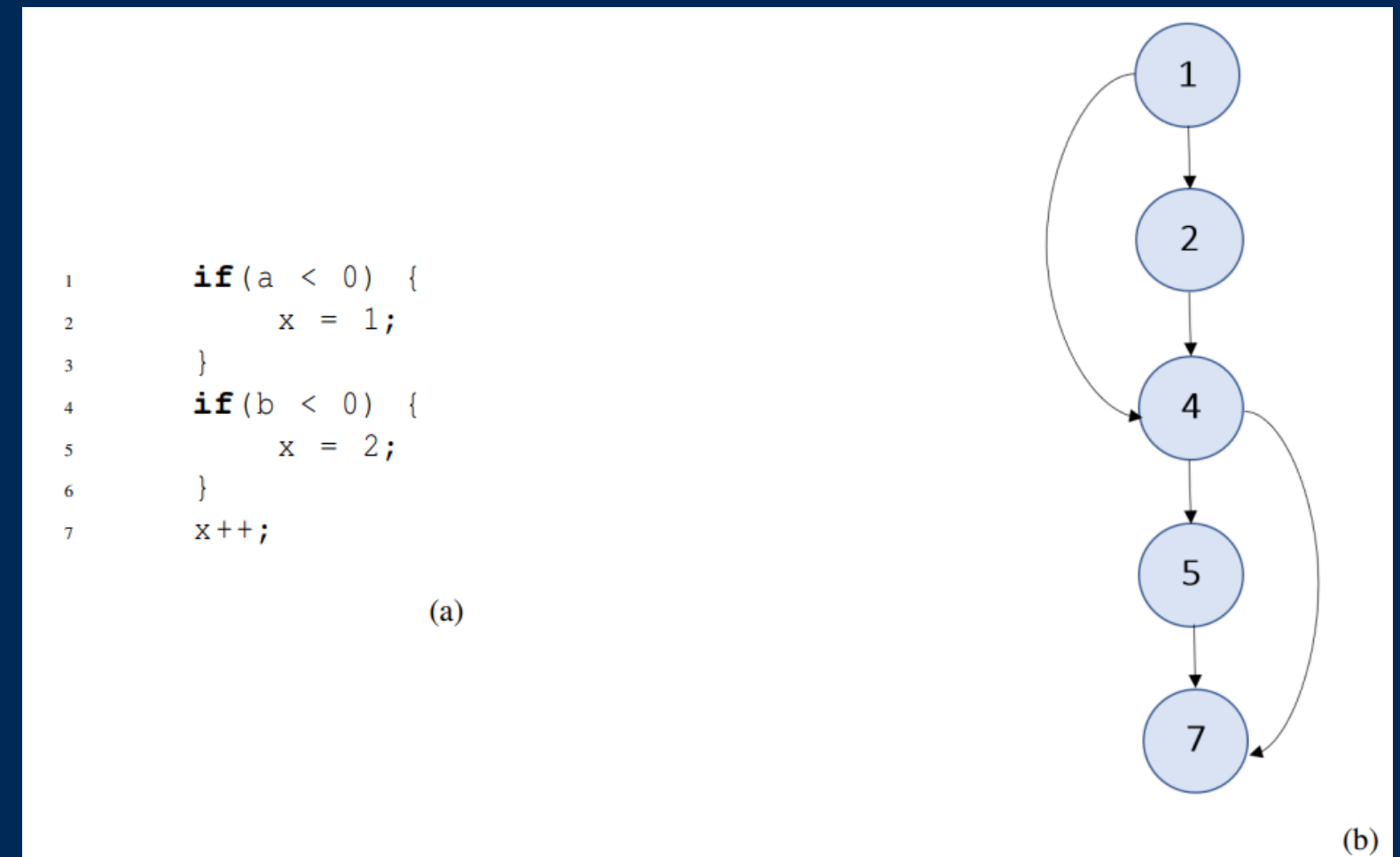
❖ Code Coverage Criteria

Code coverage testing measures the percentage of the program source code exercised based when running specific test suites.

Code coverage criteria is the requirement that a test suite needs to fulfill during software testing

- **Function Coverage** = $\frac{\text{number of functions executed}}{\text{total number of functions}} * 100\%$
- **Statement Coverage** = $\frac{\text{number of statements executed}}{\text{total number of statements}} * 100\%$
- **Branch Coverage** = $\frac{\text{number of branches executed}}{\text{total number of branches}} * 100\%$

Full coverage: any particular coverage type with 100% coverage



a = -1	b = -1
a = 1	b = 1

Background - GCC and Gcov

❖ GCC

The **GNU Compiler Collection (GCC)**, includes the front ends and libraries to support several programming languages such as C, C++, Objective-C, and etc. It needs to add [-fprofile-arcs] and [-ftest-coverage] or [--coverage] to support Gcov.

❖ GCOV

Gcov is a code coverage tool that comes as a utility from GCC and only works on code compiled with GCC . It uses two types of files, **gcno** and **gcda**, to generate coverage information that can be used to analyze various types of code coverage.

The node files with extension **gcno** produced during compilation, containing the data needed to reproduce the graph of basic blocks and the corresponding source line number to each block.

The count files with extension **gcda** produced at program termination, containing the execution count for every basic block and branch and some summary information.

Let's take `hello_world.cpp` as an example!

Source code file `hello_world.cpp` for the `hello_world` program.

```
1  #include <iostream>
2
3  int main() {
4
5      std::cout << "Hello, World!\n";
6
7      int status;
8      if(std::cout) {
9          status = 0;
10     } else {
11         status = 1;
12     }
13
14     return status;
15 }
```

Background - GCC and Gcov

1. Compile the file with `--coverage` using the GCC C++ compiler. The gcno file will be generated in this step.

```
g++ --coverage hello_world.cpp -o hello_world
```

2. Run the program generated and print the string "Hello, World!". The gcda file will be generated in this step.

```
./hello_world
```

3. Use gcov command to generate summary information and the coverage report hello_world.gcda.

```
gcov hello_world
```

Summary information of hello_world generated with gcov

```
File 'hello_world.cpp'  
Lines executed:83.33% of 6  
Creating 'hello_world.cpp.gcov'  
File '/usr/include/c++/10/iostream'  
No executable lines  
Removing 'iostream.gcov'
```

The code coverage report hello_world.cpp.gcov.

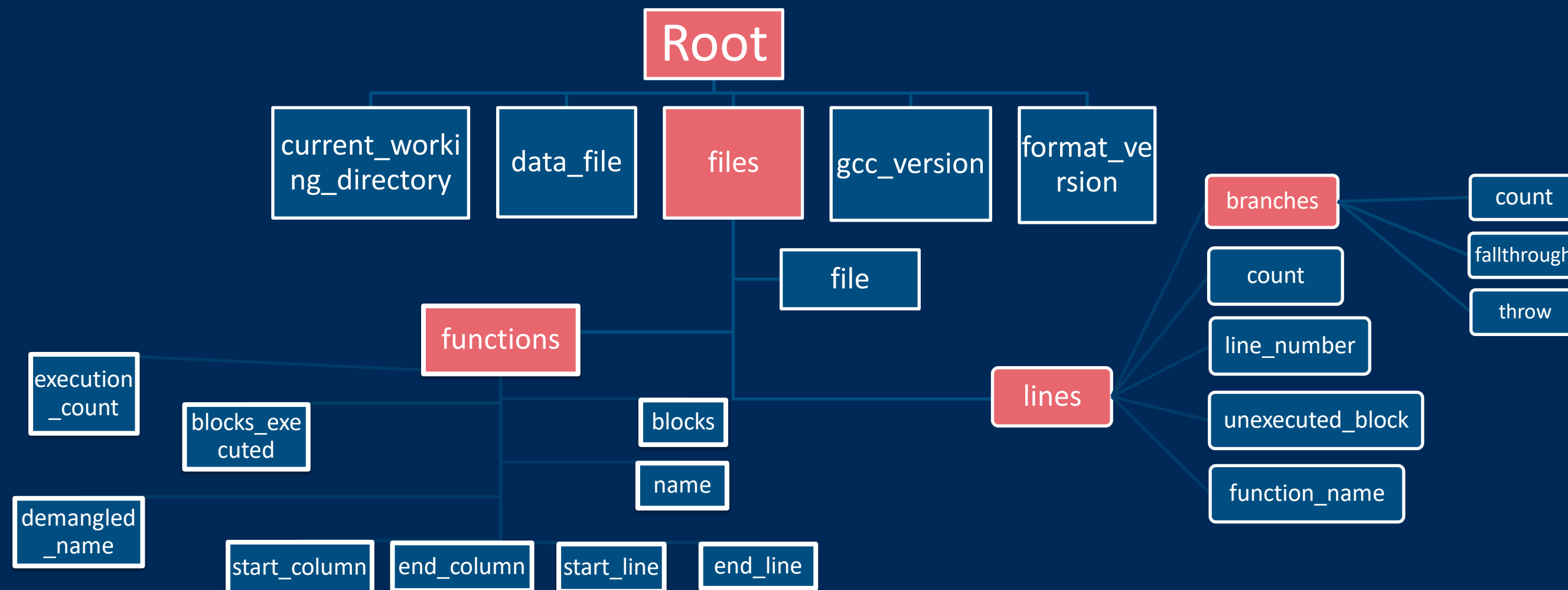
```
-: 0:Source:hello_world.cpp  
-: 0:Graph:hello_world.gcno  
-: 0:Data:hello_world.gcda  
-: 0:Runs:1  
-: 1:#include <iostream>  
-: 2:  
1: 3:int main() {  
-: 4:  
1: 5:     std::cout << "Hello, World!\n";  
-: 6:  
-: 7:     int status;  
1: 8:     if(std::cout) {  
1: 9:         status = 0;  
-: 10: } else {  
#####: 11:     status = 1;  
-: 12: }  
-: 13:  
1: 14: return status;  
-: 15:}
```


Background - GCC and Gcov

Gcov provides numerous command-line options for obtaining additional information to the coverage report.

For example: `gcov hello_world -a -b -j`

- -a or --all-blocks : adds the execution count for every basic block in the coverage report
- -b or --branch-probabilities : shows the branch frequencies in percent in the coverage report
- -j or --json-format : generates the coverage data in JSON format.



Background – Name Mangling


❖ Name Mangling

In C++, multiple functions can have the same name in the source code due to overloading or visibility within different scopes. To differentiate them when linking and compiling, the compiler using a specific set of rules assigns each function in the source code a unique name is a process known as **name mangling**.

The unique name produced by name mangling is called the **mangled name**.

The original function name in the source code is called the **demangled name**.

For example, in the Itanium application binary interface (ABI) standard for C++, the functions can be mapped as:

<code>int add (int)</code>		<code>_Z3addi</code>
<code>int add (double)</code>		<code>_Z3addd</code>

Specifically, the compiler can implement each **constructor** using multiple functions, and the same holds for **destructors**. These associated functions of a constructor or destructor typically have the same function name in the source code but different mangled name. For example, a virtual base destructor with the demangled name `base::~~base()` can have two different mangled names: `_ZN4baseD0Ev` and `_ZN4baseD2Ev`.

Software

❖ Software Introduction

The **Gcov Report Generator (GRG)** software consists of:

- A C++ header-only library to generate nicely-formatted customized code-coverage report in PDF format.
- Program called coverage to run the library through command line.

The **code coverage report generated in PDF format is formatted as:**

- The summary section includes six summary tables for function, statement, and branch coverage on per-file and per-function basis (eg: Per-File Function Coverage, Per-Func Function Coverage).
- The detail section includes the function source code and detailed coverage information (eg: execution counts of lines and branches).

The **important customized features provided by the GRG are as follow:**

- filter coverage results using file and function patterns (i.e., regular expressions);
- calculate the branch coverage with or without exceptional branches;
- format the code coverage report by keeping only the selected section contents;
- select the coverage information included by setting the function, statement, and branch coverage threshold on a per-file and per-function basis; and
- aggregate coverage results for functions with the same names in source code but different mangled names

Software

❖ The GRG Library

Application programming interface (API) of the GRG library: One primary class **Report_maker** (constructor + function call operator)
+ Two secondary classes **Global_options** and **Report_options** as input parameters.

```
1  #ifndef grg_hpp
2  #define grg_hpp
3
4  #include <cstdlib>
5  #include <boost/process.hpp> // boost::process
6  #include <memory> // std::unique_ptr
7  #include <grg/code_coverage_data_structure.hpp>
8  #include <grg/nlohmann/json.hpp> // nlohmann::json
9
10
11 namespace bp = boost::process;
12 using json = nlohmann::json;
13
14 namespace grg {
15
16     class Report_maker {
17     public:
18         // Used in the process to parse and calculate coverage
19         // statistics
20         struct Global_options {
21             bool aggregate_ctors_dtors = false;
22             bool aggregate_templates = false;
23             bool keep_regex = true;
24             std::vector<File_function_regex> filter_patterns;
25         };
26
27         // Used in the process to generate PDF coverage reports.
28         struct Report_options {
29             bool ignore_exceptional_branches = false;
30             bool function_coverage_summary = false;
31             bool statement_coverage_summary = false;
32             bool branch_coverage_summary = false;
33
34             bool function_coverage_summary_per_file = false;
35             bool function_coverage_summary_per_function = false;
36             bool statement_coverage_summary_per_file = false;
37             bool statement_coverage_summary_per_function = false;
38             bool branch_coverage_summary_per_file = false;
39             bool branch_coverage_summary_per_function = false;
40             bool detail = false;
41             bool keep_temporary_directory = false;
42             double func_cov_per_file_threshold = 101;
43             double func_cov_per_func_threshold = 101;
44             double state_cov_per_file_threshold = 101;
45             double state_cov_per_func_threshold = 101;
46             double branch_cov_per_file_threshold = 101;
47             double branch_cov_per_func_threshold = 101;
48             std::string temporary_directory_name = "";
49             std::string output_file_name = "";
50             std::string pdflatex_program = "pdflatex";
51         };
52
53         // The constructor
54         Report_maker(Global_options options,
55                     std::vector<std::string>& gcda_files);
56
57         // The function call operator
58         bool operator()(Report_options options, std::ostream& out);
59     };
60
61 } /*end of namespace grg*/
62
63 #endif /*grg_hpp*/
```

❖ The Application Program Coverage

Coverage is controlled by numerous command line options. There are two types of these options: **Global Options** and **Per-Report Options**. Global options apply to all reports generated by the program, while per-report options apply only to a single report.

Global Options

- **-c or --aggregate-constructors-destructors**

When this option is specified, coverage aggregates all of the C++ compiler-generated functions from one particular C++ constructor as if they were the same function when determining code coverage. This option also has a similar effect on destructors.

- **-t or --aggregate-templates**

When this option is specified, coverage aggregates all instantiations of the same template function as if they were the same function when determining code coverage.

- **-h or --help**

When this option is specified, coverage prints help information about using coverage to the standard output and exits without doing any further processing.

Global Options — Function Selection

- **-x REGEX or --file-pattern REGEX**

When this option is specified, coverage sets the current value of the file pattern to the extended regular expression specified by REGEX. The value of REGEX defaults to string "(.*?)" (i.e., select all files).

- **-y REGEX or --function-pattern REGEX**

When this option is specified, coverage sets the current value of the function pattern to the extended regular expression specified by REGEX. The value of REGEX defaults to string "(.*?)" (i.e., select all functions).

- **-p EXPRESSION or --pattern EXPRESSION**

When this option is specified, coverage adds a new file-function pattern to the pattern list that consists of the file-function patterns. The EXPRESSION is a string chosen from “keep” or “delete”, through which we can determine whether to keep or discard the current file-function pattern when generating the report. The value of EXPRESSION defaults to keep, which means the file-function pattern that matches the EXPRESSION will be kept.

- **-r or --reset**

When this option is specified, coverage resets all of the per-report options to their default values.

❖ The Application Program Coverage

Per-Report Options

- **-n DIRNAME or --temporary-directory-name DIRNAME**

When this option is specified, coverage sets the DIRNAME as the name of the temporary directory used to store the LaTeX source file. The directory specified must already exist. For example, "/tmp/foobar" or "apple" can be used. If this option is not specified, the system directory for temporary files will be used (i.e., the directory returned by the C++ standard library function `std::filesystem::temp_directory_path`).

- **-k or --keep-temporary-directory**

When this option is specified, coverage keeps the temporary directory that contains the LaTeX source file. If this option is not specified, the directory will be deleted.

- **-e or --ignore-exceptional-branches**

When this option is specified, coverage treats exceptional branches as if they do not exist when determining branch coverage.

- **-o PATHNAME or --output PATHNAME**

When this option is specified, coverage sets PATHNAME as the output pathname to which to write the code coverage report. If not specified, the report is printed to standard output.

- **-m PROGRAM or --pdflatex-program PROGRAM**

This option sets PROGRAM as the pathname of the program to use to convert LaTeX source to a PDF document. Any program that has a CLI compatible with pdflatex can be used. The pdflatex, lualatex, or pdflatex_frontend are possible values for PROGRAM. The value of PROGRAM defaults to pdflatex.

Per-Report Options – Report Formatting

- **-d or --detail**

When this option is specified, coverage includes the detail section in the report, which contains detailed coverage information for each selected function.

- **--function-coverage-summary-per-file/--statement-coverage-summary-per-function/--branch-coverage-summary-per-file**

When this option is specified, coverage includes the per-file function/statement/branch coverage summary table in the code coverage report.

- **--function-coverage-summary-per-function/--statement-coverage-summary-per-function/--branch-coverage-summary-per-function**

When this option is specified, coverage includes the per-function function/statement/branch coverage summary table in the code coverage report.

- **--function-coverage-summary/--statement-coverage-summary/--branch-coverage-summary**

This option works for selecting both the --function-coverage-summary-per-file and the --function-coverage-summary-per-function options (same for statement and branch).

❖ The Application Program Coverage

Per-Report Options — Function Selection

- **-F PERCENT or --func-cov-per-file-threshold PERCENT**

When this option is specified, coverage selects all functions in source files with function coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

- **-f PERCENT or --func-cov-per-func-threshold PERCENT**

When this option is specified, coverage selects all functions with function coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

- **-S PERCENT or --state-cov-per-file-threshold PERCENT**

When this option is specified, coverage selects functions in source files with statement coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

- **-s PERCENT or --state-cov-per-func-threshold PERCENT**

When this option is specified, coverage selects all functions with statement coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

- **-B PERCENT or --branch-cov-per-file-threshold PERCENT**

When this option is specified, coverage selects all functions in source files with branch coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

- **-b PERCENT or --branch-cov-per-func-threshold PERCENT**

When this option is specified, coverage selects all functions with branch coverage less than the real number PERCENT, of which the value starts from 0 and is allowed to be greater than 100. The value of the PERCENT defaults to 101. The lines in the selected function are marked in the detail section of the report.

Software

❖ Usage Example

A source file `template.cpp`:

- a template function called `bubbleSort` to sort the input array with element type specified in ascending order;
- two overloading functions with the same name `sum` used to add two input numbers but differs in the return type and input data type;
- a function called `main` that initializes some variables and calls the other functions.

Source code of the `template.cpp` used in code coverage report example

```
1  #include <iostream>
2  #include <type_traits> //std::extent
3
4  // A template function
5  template <class T>
6  void bubbleSort(T a[], int n) {
7      for (int i = 0; i < n - 1; i++) {
8          for (int j = n - 1; i < j; j--) {
9              if (a[j] < a[j - 1]) {
10                 std::swap(a[j], a[j - 1]);
11             }
12         }
13     }
14 }
15
16 // Two overloading functions
17 int sum(int a, int b) {
18     return a + b;
19 }
20
21 double sum(double a, double b) {
22     return a + b;
23 }
24
25 int main() {
26
27     // Calls template function
28     int a1[5] = {3, 5, 1, 2, 4};
29     double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
30     bubbleSort<int>(a1, std::extent<int[5]>::value);
31     bubbleSort<double>(a2, std::extent<int[5]>::value);
32
33     // Calls overload functions and outputs the results
34     int a = 1;
35     int b = 2;
36     double c = 1.2;
37     double d = 2.4;
38     std::cout << sum(a,b) << '\n';
39     std::cout << sum(c,d) << '\n';
40
41     return 0;
42 }
```


Software

❖ Usage Example

Source code for full_report.pdf that uses the GRG library API with default options



```
1  #include <iostream>
2  #include <fstream>
3  #include "grg/grg.hpp"
4
5  int main() {
6
7      // Initialize the global options
8      grg::Report_maker::Global_options global_options;
9      std::vector<std::string> gcda_files_vec;
10     std::string input_file("template.gcda");
11     gcda_files_vec.push_back(input_file);
12
13     // Call the constructor of Report_maker class
14     grg::Report_maker maker(global_options, gcda_files_vec);
15
16     // Initialize the report options
17     grg::Report_maker::Report_options report_options;
18     report_options.function_coverage_summary = true;
19     report_options.statement_coverage_summary = true;
20     report_options.branch_coverage_summary = true;
21     report_options.detail = true;
22
23     // Call the operator() to generate the coverage report stream
24     // and write it to a PDF file named full_report.pdf
25     std::fstream myfile("full_report.pdf", std::ios::out);
26     int status = maker(report_options, myfile);
27     myfile.close();
28
29     return status;
30 }
```

Source code for the file specified_report.pdf that uses the GRG library API with particular options



```
1  #include <iostream>
2  #include <fstream>
3  #include "grg/grg.hpp"
4
5  int main() {
6
7      // Initialize the global options
8      grg::Report_maker::Global_options global_options;
9      global_options.aggregate_templates = true;
10     File_function_regex cur_pattern = {".*template.cpp", "(.*?)"};
11     global_options.filter_patterns.push_back(cur_pattern);
12     std::vector<std::string> gcda_files_vec;
13     std::string input_file("template.gcda");
14     gcda_files_vec.push_back(input_file);
15
16     // Call the constructor of Report_maker class
17     grg::Report_maker maker(global_options, gcda_files_vec);
18
19     // Initialize the report options
20     grg::Report_maker::Report_options report_options;
21     report_options.function_coverage_summary = true;
22     report_options.statement_coverage_summary = true;
23     report_options.branch_coverage_summary = true;
24     report_options.detail = true;
25     double threshold = 100;
26     report_options.state_cov_per_func_threshold = threshold;
27     report_options.branch_cov_per_func_threshold = threshold;
28
29     // Call the operator() to generate the coverage report stream
30     // and write it to a PDF file named specified_report.pdf
31     std::fstream myfile("specified_report.pdf", std::ios::out);
32     int status = maker(report_options, myfile);
33     myfile.close();
34
35     return status;
36 }
```

Software

❖ Usage Example

Run coverage to generate the file full_report.pdf containing all the files and functions with default option settings:

```
./coverage --function-coverage-summary \  
--statement-coverage-summary \  
--branch-coverage-summary \  
--detail \  
--output full_report.pdf template.gcda
```

Run coverage to generate the file specified_report.pdf with specified file-function pattern and coverage thresholds:

```
./coverage --file-pattern .*template.cpp --pattern keep \  
--function-coverage-summary \  
--statement-coverage-summary \  
--branch-coverage-summary \  
--detail \  
--state-cov-per-func-threshold 100 \  
--branch-cov-per-func-threshold 100 \  
--output specified_report.pdf template.gcda
```



1 Summary

1.1 Function Coverage

1.1.1 Per-File Function Coverage

Coverage	Pathname
5/5 (100.00%)	/home/mint/Coverage/tests/template.cpp
4/4 (100.00%)	/usr/include/c++/10/bits/move.h

1.1.2 Per-Function Function Coverage

Coverage	Pathname
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main
12/12 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std::move<double&>(double&)
15/15 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std::move<int&>(int&)
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<double>>, std::is_move_constructible<double>, std::is_move_assignable<double>>::value, void>::type std::swap<double>(double&, double&)
5/5 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<int>>, std::is_move_constructible<int>, std::is_move_assignable<int>>::value, void>::type std::swap<int>(int&, int&)

1.2 Statement Coverage

1.2.1 Per-File Statement Coverage

Coverage	Pathname
33/33 (100.00%)	/home/mint/Coverage/tests/template.cpp
12/12 (100.00%)	/usr/include/c++/10/bits/move.h

1.2.2 Per-Function Statement Coverage

Coverage	Pathname
9/9 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
9/9 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
11/11 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main
2/2 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std::move<double&>(double&)
2/2 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std::move<int&>(int&)

Continued from previous page.

Coverage	Pathname
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<double>>, std::is_move_constructible<double>, std::is_move_assignable<double>>::value, void>::type std::swap<double>(double&, double&)
4/4 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<int>>, std::is_move_constructible<int>, std::is_move_assignable<int>>::value, void>::type std::swap<int>(int&, int&)

1.3 Branch Coverage

1.3.1 Per-File Branch Coverage

Coverage	Pathname
16/20 (80.00%)	/home/mint/Coverage/tests/template.cpp
0/0 (100.00%)	/usr/include/c++/10/bits/move.h

1.3.2 Per-Function Branch Coverage

Coverage	Pathname
4/8 (50.00%)	[/home/mint/Coverage/tests/template.cpp] main
6/6 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
6/6 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<int>(int*, int)
0/0 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
0/0 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<double&>::type&& std::move<double&>(double&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::remove_reference<int&>::type&& std::move<int&>(int&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<double>>, std::is_move_constructible<double>, std::is_move_assignable<double>>::value, void>::type std::swap<double>(double&, double&)
0/0 (100.00%)	[/usr/include/c++/10/bits/move.h] std::enable_if<std::__and<std::__not<std::__is_tuple_like<int>>, std::is_move_constructible<int>, std::is_move_assignable<int>>::value, void>::type std::swap<int>(int&, int&)

2 Details

2.1 /home/mint/Coverage/tests/template.cpp

void bubbleSort<double>(double*, int)

Line	Counts	Branches	Source
6	1		void bubbleSort(T a[], int n){
7	5	4:1	for (int i = 0; i < n - 1; i++){
8	14	10:4	for (int j = n - 1; i < j; j--){
9	10	4:6	if (a[j] < a[j - 1]){
10	4		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			}
15	1		}

void bubbleSort<int>(int*, int)

Line	Counts	Branches	Source
6	1		void bubbleSort(T a[], int n){
7	5	4:1	for (int i = 0; i < n - 1; i++){
8	14	10:4	for (int j = n - 1; i < j; j--){
9	10	5:5	if (a[j] < a[j - 1]){
10	5		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			}
15	1		}

sum(int, int)

Line	Counts	Branches	Source
18	1		int sum(int a, int b){
19	1		return a + b;
20			}

sum(double, double)

Line	Counts	Branches	Source
22	1		double sum(double a, double b){
23	1		return a + b;
24			}

main

Line	Counts	Branches	Source
27	1		int main(){
28			
29			// Calls template function
30	1		int a1[5] = {3, 5, 1, 2, 4};
31	1		double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
32	1		bubbleSort<int>(a1, std::extent<int[5]>::value);
33	1		bubbleSort<double>(a2, std::extent<int[5]>::value);
34			
35			// Calls overload functions and outputs the results
36	1		int a = 1;
37	1		int b = 2;
38	1		double c = 1.2;
39	1		double d = 2.4;
40	1	1:0:1:0	std::cout << sum(a,b) << '\n';
41	1	1:0:1:0	std::cout << sum(c,d) << '\n';
42			
43	1		return 0;
44			}

2.2 /usr/include/c++/10/bits/move.h

std::remove_reference<double&>::type&& std::move<double&>(double&)

Line	Counts	Branches	Source
101	12		move(_Tp&& __t)noexcept
102	12		{ return static_cast<typename std::remove_reference<_Tp>::type&&(&__t); }

std::remove_reference<int&>::type&& std::move<int&>(int&)

Line	Counts	Branches	Source
101	15		move(_Tp&& __t)noexcept
102	15		{ return static_cast<typename std::remove_reference<_Tp>::type&&(&__t); }



❖ Usage Example

The file specified_report.pdf generated the by GRG software.

1 Summary

1.1 Function Coverage

1.1.1 Per-File Function Coverage

Coverage	Pathname
4/4 (100.00%)	/home/mint/Coverage/tests/template.cpp

1.1.2 Per-Function Function Coverage

Coverage	Pathname
2/2 (100.00%)	[/home/mint/Coverage/tests/template.cpp] void bubbleSort<double>(double*, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(int, int)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] sum(double, double)
1/1 (100.00%)	[/home/mint/Coverage/tests/template.cpp] main

1.2 Statement Coverage

1.2.1 Per-File Statement Coverage

Coverage	Pathname
24/24 (100.00%)	/home/mint/Coverage/tests/template.cpp

1.2.2 Per-Function Statement Coverage

1.3 Branch Coverage

1.3.1 Per-File Branch Coverage

Coverage	Pathname
10/14 (71.43%)	/home/mint/Coverage/tests/template.cpp

1.3.2 Per-Function Branch Coverage

Coverage	Pathname
4/8 (50.00%)	[/home/mint/Coverage/tests/template.cpp] main

2 Details

2.1 /home/mint/Coverage/tests/template.cpp

void bubbleSort<double>(double*, int)

Line	Counts	Branches	Source
6	2		void bubbleSort(T a[], int n){
7	10	8:2	for (int i = 0; i < n - 1; i++){
8	28	20:8	for (int j = n - 1; i < j; j--){
9	20	9:11	if (a[j] < a[j - 1]){
10	9		std::swap(a[j], a[j - 1]);
11			}
12			}
13			}
14			}
15	2		}

sum(int, int)

Line	Counts	Branches	Source
18	1		int sum(int a, int b){
19	1		return a + b;
20			}

sum(double, double)

Line	Counts	Branches	Source
22	1		double sum(double a, double b){
23	1		return a + b;
24			}

main

Line	Counts	Branches	Source
27	1		int main(){
28			
29			// Calls template function
30	1		int a1[5] = {3, 5, 1, 2, 4};
31	1		double a2[5] = {2.3, 5.6, 1.1, 2.2, 10.0};
32	1		bubbleSort<int>(a1, std::extent<int[5]>::value);
33	1		bubbleSort<double>(a2, std::extent<int[5]>::value);
34			
35			// Calls overload functions and outputs the results
36	1		int a = 1;
37	1		int b = 2;
38	1		double c = 1.2;
39	1		double d = 2.4;
40	1	1:0:1:0	std::cout << sum(a,b) << '\n';
41	1	1:0:1:0	std::cout << sum(c,d) << '\n';
42			
43	1		return 0;
44			}

Software

❖ Software Installation

The following versions of required software packages have been verified to work with the GRG software:

- GCC 10.3.0;
- Boost 1.71.0; and
- pdfTeX 1.40.22 (TeX Live 2021).

The GRG software uses CMake for compiling, building, testing, and installation:

1. Generate the native build files by running the command:

```
cmake -H$SOURCE_DIR -B$BUILD_DIR
```

2. Build the GRG software and test files by running the command:

```
cmake --build $BUILD_DIR
```

3. Test the GRG software built by running the command follows:

```
cmake --build $BUILD_DIR --target test
```

4. Install the GRG software by running the command:

```
cmake --build $BUILD_DIR --target install
```

The text like the following will be printed to the standard output upon successful completion of the tests, and the code coverage reports will be presented in the directory named CoverageTestResults under the \$SOURCE_DIR directory.

```
100% tests passed , 0 tests failed out of 4
```

Conclusion and Future Work

- The importance of software testing and the criteria of structural coverage analysis have been studied.
- The GRG software developed based on Gcov containing a library and a program called coverage.
- The GRG software can generate customized code coverage report in PDF format for function, statement, and branch coverage on per-file and per-function basis.
- The GRG software provides rich functionalities, including filtering functions using regular expressions, calculating the branch coverage with or without exceptional branches, selecting the report contents by setting coverage threshold, and aggregating associated functions with different mangled names.
- In the future, the GRG software could allow users to output coverage statistics into a plain text file.
- In addition, the GRG software may need to add mangled function names when presenting the report.

Thank you!
Any Questions?