DIVA

Debug Information Visual Analyzer

User guide

Version 0.4

Contents

1 Introduction 4
1.1 Use-case 1: inspecting the debug information 4
1.2 Use-case 2: comparing the debug information from two compulations 4
1.3 Use-case 3: Using DIVA outputs for regression testing 5
2 Basic functionality 5
2.1 Helloworld 5
2.2 The DIVA output 6
2.3 Comparing DIVA outputs 7
3 User options 10
3.1 Summary of options 10
3.2 Basic options 12
3.3 More command line options 16
4 DIVA advanced options 20
4.1 Printing DWARF offsets and tag information 20
4.2 Sorting based on the DWARF offset 21
4.3 Advanced command line options 21
4.4 DWARF warning options 30
5 DIVA object output format 30
5.1 Textual output format 30
5.2 YAML output format 34
6 Appendix 39
6.1 Roadmap 39
6.2 Known issues 39
6.3 Error messages 39
6.4 Glossary of terms 40

1 Introduction

DIVA (Debug Information Visual Analyzer) is a command line tool that processes DWARF[1] debugging information contained within ELF[2] files. DIVA prints a high-level representation of the debugging information and is designed to be readable by high-level language programmers (i.e. C/C++, Fortran programmers who are not compiler or DWARF experts).

DIVA's high-level representation will be consistent regardless of how the debug information is arrange or formatted within the ELF and can be used to inspect the debug information, to check that it correctly represents the original program source code. As such, DIVA can be used to:

a. compare the debug information generated from multiple compilers, or different versions of the same compiler, and
b. compare the debug information generated from different compiler switches, or different DWARF specifications (i.e. DWARF 2,3,4 and 5).

DIVA can print it's high-level representation as YAML and in a textual form, the former is useful for scripting (such as python-based QA test) and the latter is useful for manual inspection/reading. The DIVA examples given in this user guide are printed in the textual form for compactness. Figure 1 shows the DIVA textual output from the HelloWorld example and it can be read as: a function 'main' that returns an 'int' is declared on line 5.

```
$ cd <path to>\diva\examples\
$ clang++ helloworld.cpp –c –g –o helloworld.o
$ diva helloworld.o

           {InputFile} "helloworld.o"

             {CompileUnit} "helloworld.cpp"

  {Source} "helloworld.cpp"
     5          {Function} "main" -> "int"
```

Figure 1. DIVA output of HelloWorld.o

1.1 Use-case 1: inspecting the debug information

Given only one ELF object file (such as 'helloworld.o') DIVA will allowing you to inspect what debug information is present. This output can be compared with the source code to check whether there are some parts missing or unexpectedly present. The advanced command line options allow you to specify exactly which DIVA objects to print, alternatively, with no options, DIVA will print the common objects (as it's default output), such as {Funciton}, {Class} and {Namespace}. For a full list of DIVA objects, refer to section 3.1.

1.2 Use-case 2: comparing the debug information from two compulations

When more than one ELF object file is given at the command line, DIVA compares the logical representation from the first ELF with all subsequent ELF object files. When you have an ELF object file with "good" debug information, this can be compared against one, or more, other ELF files generated from the same source file(s) to determine whether anything has changed. For more information on comparing ELF files, see section 2.3.

[1] DWARF DWARF is a widely used, standardized debugging data format. DWARF was originally designed along with Executable and Linkable Format (ELF), although it is independent of object file formats. The name is a medieval fantasy complement to "ELF" that has no official meaning, although the backronym 'Debugging With Attributed Record Formats' was later proposed.

[2] ELF ELF is the Executable and Linkable Format and is a common standard file format for executables, object code, shared libraries, and core dumps.

```
$ cd <path to>\diva\examples\
$ clang++ helloworld.cpp –c –g –O0 –o helloworld_O0.o
$ clang++ helloworld.cpp –c –g –O2 –o helloworld_O2.o
$ diva helloworld_O0.o helloworld_O2.o --show-all
```

1.3 Use-case 3: Using DIVA outputs for regression testing

DIVA can output the high-level representation in a YAML format that can then be used as an input to python scripts to read and process. Regression tests can be written around the DIVA generated YAML to validate the debug information.

2 Basic functionality

DIVA is available under both Window and Linux, and is distributed with no installation program with three directories: \bin, \doc, and \examples. To start, copy the latest DIVA version to your preferred location and add the \bin path into your PATH environment variable. All of the code (with pre-compiled ELF object files) used in this document can be found in the \examples directory. These examples can be compiled with any Clang or GNU toolchain. Throughout this document, clang++ is used in the examples, this can be substituted with g++ to gain a similar output.

This user guide only assumes the following:

- Familiarity with the C/C++ programming language
- Access to a compiler that emits DWARF Debugging Information in an ELF format
- Access to the DIVA binary and examples

Compiler and DWARF knowledge is not assumed, but would be of advantage for the advanced options in section 4.

2.1 Helloworld

To follow convention, we will use a "Hello World!" program to show you how to start. First compile the \examples\HelloWorld.cpp with the compiler -g option to create an object file with DWARF debugging information (although a precompiled object file is provided in the release). Then invoke DIVA on the generated helloworld.o file without any options to view the default DIVA output.

In Figure 1 you can see the list of default DIVA objects: {InputFile}, {CompileUnit} and {Function}. Adding the --show-all option will include more DIVA objects and attributes, such as: {PrimitiveTypes} as seen in Figure 3. Each DIVA object may have one or many attributes, such as class inheritance, where attributes will only be printed when the debug information contained in the input ELF file has associated the attribute with the object.

```
1 /* HelloWorld.cpp */
2
3 #include <stdio.h>
4
5 int main()
6 {
7   printf("Hello, World\n");
8   return 0;
9 }
```

Figure 2. HelloWorld.cpp

```
$ clang++ -c -g helloworld.cpp -o helloworld.o
$ diva helloworld.o --show-all

           {InputFile} "helloworld.o"

             {CompileUnit} "helloworld.cpp"
               {PrimitiveType} -> "int"
                   - 4 bytes

  {Source} "helloworld.cpp"
     5         {Function} "main" -> "int"
```

Figure 3. The default DIVA output for HelloWorld.o

2.2 The DIVA output

The DIVA output contains a tree structure of DIVA objects. The name of each DIVA object is printed within the {} braces and is immediately followed by the object's content and these objects correspond to many of the constructs in the C/C++ language; such as: File, Function, Line, Namespace, etc.. The command line options control which kind of object and which attribute of each object is printed. Refer to section 3 for a full list of command line user options.

The primary use of DIVA is to show/compare the high-level view of the debug information. It is anticipated that most users will only print out the DIVA {} objects; however for users who want to see DWARF information there is some (but limited) capability to print the DWARF information. All DWARF specific information will be printed inside [] braces. For more information about DIVA's DWARF printing capabilities, refer to section 4.

2.2.1 Layout of the textual output

By default, DIVA prints the output in three columns, the first columns lists any DIVA object flags or DWARF tags, the second column lists the line position with reference to the source file, and the third column lists the DIVA objects. The DIVA objects are printed following the format given in Figure 4, where a each line either begins with a kind name in {} braces or a '-' that denotes an attribute of the preceeding kind. Each object is printed with the same indent when they are part of the same lexical block and each lexical block is indented from its parent block. The specifics of each DIVA object's textual structure is described in section 5.1.

```
{<DIVA object>} "<name of instance>" -> "<type>"
    - <attribute details>
    - <attribute details>
```

Figure 4. Structure of a single DIVA object that represents the debug information in a textual format

2.2.2 Layout of the YAML output

DIVA can print the same information given in the textual output as YAML. Each instance of a DIVA object will follow the form described in Figure 5. Each instance will contain "object", "name", "type", "souce", "dwarf", "attributes" and "children" fields, regardless of whether that field contains any information. The specifics of each DIVA object's YAML structure is described in section 5.2.

```
-  object: "Name of DIVA object"
   name: "<name of instance>"
   type: "<type>"
   source:
       line: <line number>
       file: "<file path>"
   dwarf:
       offset: <dwarf offset>
       tag: "<dwarf tag>"
   attributes: {}
   children: []
```

Figure 5. DIVA debug information in YAML format

2.2.3 Printing to a file

DIVA prints, by default, to stdout which can be sent to a file by the standard redirection mechanism (> and >>). However, using the --output-dir option, DIVA will direct its output to files in a directory spliting each {CompileUnit} into a separate file. This can be useful when the input ELF file represents a large application for which the DIVA output can be many hundreds of megabytes in size.

For the helloworld example given in Figure 6 a directory *helloworld_diva* is created in the same directory as the helloWorld.o file and it will contain a *helloworld_cpp.txt* file with the DIVA textual output. If the compilation unit name includes any '/' or '\' as part of a directory structure, they are replaced with '_' in order to have a flat name.

```
$ cd <path to>\diva\examples\
$ clang++ helloworld.cpp –c –g –o helloworld.o
$ diva helloworld.o --output-dir=helloworld_diva
$ dir /b /ad
helloworld_diva
```

Figure 6. Commands for printing each {CompileUnit} to a separate file

2.3 **Comparing DIVA outputs**

A common toolchain development task is to evaluate whether different versions of the same compiler are emitting the equivalent debugging information. Compilers can emit debug information that is structurally laid out differently but are functionally the same; this makes it difficult to compare low-level debug information generated from the same source file. Comparisons using DIVA's high-level representation are straight forward and can be used to find *missing, added or changed* debug information. To illustrate the concept of DIVA output comparison we will compare \examples\scopes.cpp compiled with and without -D_A; this will generate two object files, one with INT defined globally (*scopes_org.o*) and one with INT defined locally to foo() (*scopes_mod.o*). Generating the DIVA output for both object files and redirect them to individual text files, *scopes_org.txt* and *scopes_mod.txt*, we can compare them using any diff (difference) tool.

```
$ clang++ -c -w -g -O0 scopes.cpp -o scopes_org.o -D_A
$ clang++ -c -w -g -O0 scopes.cpp -o scopes_mod.o -U_A
$ diva --show-all scopes_org.o > scopes_org.txt
$ diva --show-all scopes_mod.o > scopes_mod.txt
$ diff_tool scopes_org.txt scopes_mod.txt

 1 /* Scopes.cpp */
 2 #ifdef _A
 3 typedef int INT;
 4 #endif /* _A */
 5
 6 void foo()
 7 {
 8 #ifndef _A
 9   typedef int INT;
10 #endif /* _A */
11   INT a = 1;
12 }
```

Figure 7. Example that forces a difference in the debug information

2.3.1 Comparison using a *diff* tool

A standard *diff* (difference) tool can be used to compare two textual DIVA output (via files) to find differences in generated debug information. Figure 13 shows the output from a generic diff tool comparing the DIVA output from a release compiler (shown on the left hand side panel) with the DIVA output from a new compiler (shown on the right hand side). The diff report shows that the {Alias} object is located in differing lexical blocks.

If the input file represents a complete application you may want to split each compilation unit's DIVA view into individual files (--output-dir). A *diff* tool can visualize each file contained within the output directory separately.

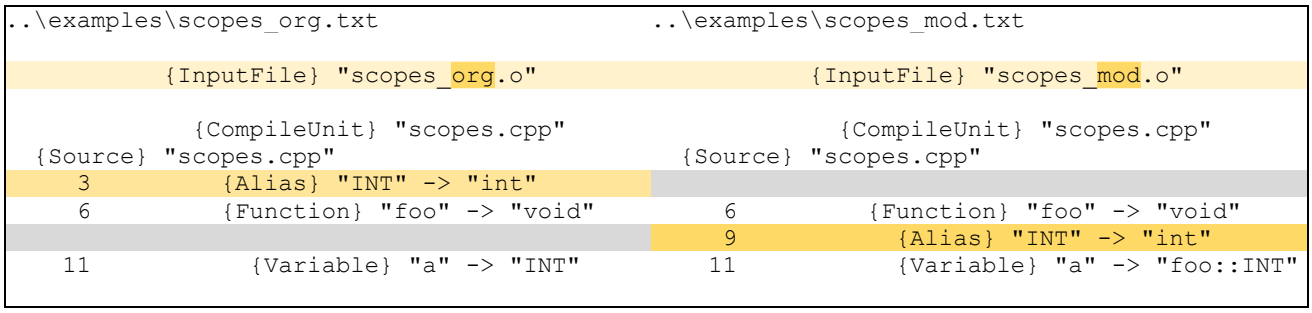

Figure 8 Comparing two DIVA outputs using a graphical *diff* tool

2.3.2 Built-in comparison module

When more than one file is specified on the command line, DIVA will switch into a comparison mode. The first input ELF object file will become a *reference* and all subsequent input ELF object files will be compared against that reference.

Typically, the reference would be known to contain 'good' debug information (that has previously been triaged) and you want to find differences in the subsequent ELF object files, such as ELF object files that were generated from the same source file(s) using new builds of a compiler. DIVA has introduced a 'golden' attribute[3] that allows you to define which parts of the reference ELF object file to be included/excluded in the comparison. By default, all DIVA objects are initially marked as golden. As a result of triage, individual DIVA objects can subsequently be marked as not-golden so that it will be ignored during future DIVA comparisons. Alternatively, the --filter option can be used to filter out any DIVA object that is not required in a comparison.

```
$ diva scopes_org.o scopes_mod.o --show-summary

Reference: scopes_org.o
Target:    scopes_mod.o

Missing {Alias} 'INT' at line 3
Missing {Function} 'foo' at line 6

---------------------------------------
Object    Expected   Missing     Added
---------------------------------------
Scopes           3         1         0
Symbols          1         0         0
Lines            4         0         0
Types            2         1         0
---------------------------------------
Total           10         2         0
```

Figure 9 DIVA comparison summary report

The summary report includes the *expected*, *missing* and *added* DIVA objects. This difference summary table can be useful for automated testing when only difference is interesting. If more comparison information is needed, use the '--show-all' option to give a more detailed report that will include a description for the *missing, added* DIVA objects. Figure 10 gives an example of DIVA's full comparison report describing that the type definition 'INT' is missing and the output includes the location for the missing DIVA object.

[3]Comparisons based on the 'golden' attribute is not supported in the initial releases of DIVA.

```
$ diva scopes_org.o scopes_mod.o --show-all

Reference: scopes_org.o
Target:    scopes_mod.o

Missing {Alias} 'INT' at line 3
      {InputFile} 'scopes_org.o'
        {CompileUnit} 'scopes.cpp'
    2       {Alias} INT

Missing {Function} 'foo' at line 6
       {InputFile} 'scopes_org.o'
         {CompileUnit} 'scopes.cpp'
    4       {Function} foo
```

Figure 10 DIVA comparison full report

2.3.3 Default comparison settings

Structural comparisons of the DIVA representation are always performed regardless of the command line options. These comparisons ensure that two similar DIVA objects are not incorrectly considered to be the same, causing incorrect comparison results.

2.3.4 Gold degrades

The debug information within an ELF file may initially be deemed as 'good' and is used as a reference, subsequent ELF files (from new compiler versions) can then be compared to it. However, over time, triage results may prove that the parts of the debug information in the 'good' ELF file to be incorrect. There are now two options: (1) create a new reference, or (2) mark the invalid objects in the reference so that they are ignored in future comparison results.

DIVA calls all of the 'good' objects in a reference "golden" and objects that are in-valid (or should be ignored in any comparison) as "non-golden". Everything in a reference ELF is deemed to be golden until it is flagged to be non-golden. The --show-golden command line option will print a "G" against each object in the output to show the current state.

2.3.5 How to determine goldenness[4]

Option 1 – Manual triage of ELF reference candidates

This starts with the assumption that all objects are valid (i.e. golden). Someone then has to manually triage the DIVA output to assess whether it is correct and then mark those objects that are invalid to be non-golden.

Option 2 – Use the intersection of many toolchains

We could define DIVA objects as golden when they are consistent across a number of compilers, this can be different revisions of the same compiler or across different compiler technologies. This method does not determine whether the golden objects are correct; however, it can be useful to quickly determine whether a compiler patch has made an impact on the debug information.

Option 3 – Ignore specific objects

Some regression tests may want to focus on specific parts of the debug information, or specific DIVA objects, such as {Namespace}. A series of regression tests may be created to validate the correctness of namespaces in the ELF debug information. As such, references can be generates with various structures of namespaces with only the {Namespace} objects flagged as goldern. This would mean any subsequent DIVA comparison will only examine the {Namespace} objects.

[4] In the initial releases of DIVA, there is no mechanism for marking objects as non-golden.

3 User options

The DIVA command line options control the format (textual/YAML) of the printed output and what objects are printed. DIVA's default output (i.e. no command line options) is designed to print the most common objects from a high-level language programmer's point of view in the textual format.

Generally, the effect of all command line options concatenate meaning the side-effect of an option will add to the side-effect of all the preceding options. For example, if both --show-inputfile and --show-codeline are specified, both {Source} filenames and {CodeLines} are printed in the DIVA output. Some options can be used multiple times to concatenated their affect, such as --filter, however others, such as --sort, cannot be concatenated and the last instance will take precedence. Lastly, some options may override the side-effect of options that preceed it. For example, if --show-inputfile, --show-none and --show-codeline are specified, only code lines will be printed because the --show-none will override the --show-inputfile option. The {CodeLine}s will be printed because --show-codeline is listed after the --show-none.

Each "show" option that controls the printing of a specific DIVA object has a corresponding 'no' option to disable it, such as --show-codeline and --no-show-codeline.

Any term documented within <> braces is expected to be substituted by an appropriate term.

For the advanced users who want to print DWARF offsets and tags, these are always printed inside [] braces.

3.1 **Summary of options**

3.1.1 Command line options

Using 'diva --help', the following command line options are listed. Each of these options are detailed in sections 3.2 through 3.3.

```
General options
  -h  --help                  Display basic help information
      --help-more             Display more option information
      --help-advanced         Display advanced option information
  -v  --version               Display the version information
  -q  --quiet                 Suppress output to stdout


Output options
  -a  --show-all              Print all (expect advanced) objects and attributes
  -b  --show-brief            Print all common objects and attributes (default)
  -t  --show-summary          Print the summary table
  -d  --output-dir[=<dir>]    Print the output into a directory with each compile
                              unit's output in a separate file. If no dir is given,
                              then diva will use the input_file string to create an
                              output directory.
      --output=<text|yaml>    A comma separated list of output formats. Available
                              formats include: 'text', 'yaml'.
Sort options
      --sort=<key>            Primary key used when ordering the output objects
                              within the same block. By default the primary key
                              is "line" and the secondary key is the object name.
                              Possible keys include: 'line','offset','name'
```

```
Filter options
      --filter [any=]<text>    Only show objects with <text> as its instance name.
                               The output will only include exact matches unless
                               "any=" is used. Regex is accepted.
                               Multiple filters are accepted to show objects that match
                               either. Example, to show objects with either "Hello" or
                               "World" in the object instance name, use:
                               --filter any="Hello",any="World" or
                               --filter any="Hello" --filter any="World"
      --tree [any=]<text>      Same as --filter, except the whole subtree of any
                               matching object will printed.
```

3.1.2 Extended command line options

Using ‘diva --help-more’ additional show options are listed which allow you to control which DIVA objects to print and hide; each of these options are detailed in section 3.3. The --show-none option can be used to hide all default objects allowing individual objects to be added to the output.

To hide a specific object, insert 'no-' after the '--'. If two or more options conflict, prescience is given to the latter of the conflicting options.

Example:

- Using "`diva --show-block --no-show-block`", DIVA will not print any block objects.
- Using "`diva --show-all --no-show-block`", DIVA will show all common object except blocks.
- Using "`diva --show-none --show-block`", DIVA will only show blocks.

In the list below, the options labeled with a star (*) are enabled by default or when "--show-brief" is given and all of these options are enabled when "--show-all" is given.

```
    --show-alias               Print alias*
    --show-block               Print blocks*
    --show-block-attributes    Print block attributes (e.g. try,catch)
    --show-class               Print classes*
    --show-enum                Print enums*
    --show-function            Print functions*
    --show-member              Print class members*
    --show-namespace           Print namespaces*
    --show-parameter           Print parameters*
    --show-primitivetype       Print primitives
    --show-struct              Print structures*
    --show-template            Print templates*
    --show-union               Print unions*
    --show-using               Print using instances*
    --show-variable            Print variables*
```

3.1.3 Advanced command line options

The following options can be used to add additional (advanced) attributes and flags to the output. These options are only considered useful for compiler-developers who are working at the DWARF level, and these options are not enabled by "--show-brief" or "--show-all".

```
To disable an option, insert 'no-' after the '--'.

      --show-codeline             Print code lines
      --show-codeline-attributes  Print code line attributes (NewStatement, PrologueEnd)
      --show-combined             Print combined scope attributes
      --show-DWARF-offset         Print DWARF offsets
      --show-DWARF-parent         Print parent object DWARF offsets
      --show-DWARF-tag            Print DWARF tag attribute
      --show-generated            Print compiler generated attributes
      --show-global               Print "global" attributes
      --show-golden               Print "golden" attributes
      --show-indent               Print indentations to reflect context (default on)
      --show-level                Print lexical block levels
      --show-only-globals         Print only "global" objects
      --show-only-locals          Print only "local" objects
      --show-qualified            Print "qualified name" attributes
      --show-underlying           Print underlying type attributes
      --show-void                 Print "void" type attributes (default on)
      --show-zero                 Print zero line number attributes
```

3.2 Basic options

3.2.1 Output options

-a
--show-all

With the --show-all option, the DIVA output will include all standard DIVA objects and all standard attributes. There are some additional advanced DIVA outputs that are considered only relevant for really low level compiler engineers - see section 4.3.

Example: Showing all DIVA objects

```
$ diva example_09.o --show-all

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"
               {PrimitiveType} -> "char"
                   - 1 bytes
               {PrimitiveType} -> "int"
                   - 4 bytes

  {Source} "example_09.cpp"
     2         {Alias} "CHAR" -> "char"
     4         {Class} "A"
     5           {Member} private "a" -> "int"
     8         {Variable} "a" -> "A"
    10         {Function} "foo" -> "CHAR"
    10           {Parameter} "p" -> "char *"
    12           {Variable} "c" -> "CHAR"
```

-b
--show-brief

With the --show-brief option, the DIVA output will include most of the standard DIVA objects and most of the standard attributes. The show brief does not output {PrimitiveType} objects, {Block} or {CodeLine} attributes nor the variable, parameter or member location coverage attributes.

The --show-brief is enabled by default. To disable it, use --show-none.

Example: Showing all DIVA objects

```
$ diva example_09.o --show-brief

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     2         {Alias} "CHAR" -> "char"
     4         {Class} "A"
     5           {Member} private "a" -> "int"
     8         {Variable} "a" -> "A"
    10         {Function} "foo" -> "CHAR"
    10           {Parameter} "p" -> "char *"
    12           {Variable} "c" -> "CHAR"
```

-t
--show-summary

With the --show-summary option, a summary table is included at the end of the DIVA output listing both the total number of DIVA objects and the number of DIVA objects in the DIVA output.

Example: A DIVA output followed by the summary table

```
$ diva example_01.o --show-summary

           {InputFile} "example_01.o"

             {CompileUnit} "example_01.cpp"

  {Source} "example_01.cpp"
     2         {Function} "foo" -> "void"
     2           {Parameter} "c" -> "char"
     4           {Variable} "i" -> "int"

-----------------------------
Object       Total    Printed
-----------------------------
Scopes           3          3
Symbols          2          2
Types            2          0
Lines            4          0
-----------------------------
Total           11          5
```

-d
--output-dir

The --output-dir option creates files containing the DIVA output, one for each {CompilerUnit} in the directory specific or in a default (<inputfile name>_diva) directory that will be put into the same directory as the input file. The output files will contain the DIVA output in the format specific by --output, where if more than one format is specific, a file will be created for each format.

Example: DIVA output printed into one file per {CompileUnit}

```
$ diva example_16.elf --output-dir=example_16_elf
```

Example: File output listing

```
$ dir example_16_elf /b

Directory of ..\examples\example_16_elf

example_16_cpp.txt
example_16_global_cpp.txt
example_16_local_cpp.txt
```

3.2.2 Sort option

-S
--sort

The sort option changes the order in which the DIVA objects are printed within its block. By default the DIVA objects are sorted by line and this layout will closely follow the flow

of the original C/C++ source code. However, to give more flexibility when processing DIVA output, the DIVA objects can be sorted by other properties. A primary key is to follow the sort option. The sort will only affect the order the objects within the same block.

```
-S=<line|offset|name>
--sort=<line|offset|name>
```

Examples: Using the following sample C++ code:

```
 1 /* Example_09.cpp */
 2 typedef char CHAR;
 3
 4 class A {
 5   int a;
 6 };
 7
 8 A a;
 9
10 CHAR foo(char *p)
11 {
12   CHAR c = *p;
13   return c;
14 }
```

Example: Sort the DIVA output by line

```
$ diva example_09.o --sort=line

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     2         {Alias} "CHAR" -> "char"
     4         {Class} "A"
     5           {Member} private "a" -> "int"
     8         {Variable} "a" -> "A"
    10         {Function} "foo" -> "CHAR"
    10           {Parameter} "p" -> "char *"
    12           {Variable} "c" -> "CHAR"
```

Example: Sort the DIVA output by name - this will ascendingly-alpherbetically order each line within each block according to its name, where a block is shown by its indent and objects with no name will appear at the top.

```
$ diva example_09.o --sort=name

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     4         {Class} "A"
     5           {Member} private "a" -> "int"
     2         {Alias} "CHAR" -> "char"
     8         {Variable} "a" -> "A"
    10         {Function} "foo" -> "CHAR"
    12           {Variable} "c" -> "CHAR"
    10           {Parameter} "p" -> "char *"
```

3.2.3 Filter option

--filter
--tree

The filtering options hide all DIVA objects in the output that does not match <text>. The given <text> can be a regular expression and 'any=' can be used for partial matching where the default is an exact match. Multiple filtering options can be specified and their affect is

concatenated, however --filter and --tree cannot be concatenated, where --filter takes precedence.

```
--filter [any=]<text>
--tree [any=]<text>
```

Option	Filtering Action	Note
any=<text>	Check if <text> is contained in the 'name of the object instance'. In the case of objects that are lines, its number is used.	
<text>	Check if <text> is an exact match to the 'name of the object instance. In the case of objects that are lines, its number is used.	Default

Table 1 - Filtering options settings

Examples: Using the following sample C++ code:

```
 1 /* Example_09.cpp */
 2 typedef char CHAR;
 3
 4 class A {
 5   int a;
 6 };
 7
 8 A a;
 9
10 CHAR foo(char *p)
11 {
12   CHAR c = *p;
13   return c;
14 }
```

Example: unfiltered

```
$ diva example_09.o

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     2          {Alias} "CHAR" -> "char"
     4          {Class} "A"
     5            {Member} private "a" -> "int"
     8          {Variable} "a" -> "A"
    10          {Function} "foo" -> "CHAR"
    10            {Parameter} "p" -> "char *"
    12            {Variable} "c" -> "CHAR"
```

Example: Concatentated filter operation that shows all object names matching 'a' and 'p'

```
$ diva example_09.o --filter=a --filter=p

           {InputFile} "example_09.o"

  {Source} "example_09.cpp"
     5            {Member} private "a" -> "int"
     8          {Variable} "a" -> "A"
    10            {Parameter} "p" -> "* char"
```

Example: Filter for the symbols called a or p (using a regular expression)

```
$ diva example_09.o --filter="(a|p)"

           {InputFile} "example_09.o"

  {Source} "example_09.cpp"
     5           {Member} private "a" -> "int"
     8         {Variable} "a" -> "A"
    10           {Parameter} "p" -> "char *"
```

Example: Filter for the lines numbered 12

```
$ diva example_09.o --filter=12

   {InputFile} "example_09.o"
     {CompileUnit} "example_09.cpp"

{Source} "example_09.cpp"
  12       {Variable} "c" -> "CHAR"
```

Example: Showing only the lexical trees for DIVA objects called foo

```
$ diva example_09.o --tree=foo

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
    10         {Function} "foo" -> "CHAR"
    10           {Parameter} "p" -> "char *"
    12           {Variable} "c" -> "CHAR"
```

3.3 More command line options

--show-alias

Show (or hide) the alias objects, which describe a typedef or template alias.

Example: Showing DIVA alias objects

```
$ diva example_03.o --show-alias

           {InputFile} "example_03.o"

             {CompileUnit} "example_03.cpp"

  {Source} "example_03.cpp"
     2         {Alias} "INTEGER" -> "int"
     4         {Function} "foo" -> "void"
     4           {Parameter} "c" -> "char"
     6           {Variable} "i" -> "INTEGER"
```

Example: Hiding DIVA alias objects

```
$ diva example_03.o --no-show-alias

           {InputFile} "example_03.o"

             {CompileUnit} "example_03.cpp"

  {Source} "example_03.cpp"
     4         {Function} "foo" -> "void"
     4           {Parameter} "c" -> "char"
     6           {Variable} "i" -> "INTEGER"
```

--show-block

Show (or hide) the block objects, which describe a lexical block.

Example: Show DIVA block objects

```
$ diva example_13.o --show-block

           {InputFile} "example_13.o"

             {CompileUnit} "example_13.cpp"

  {Source} "example_13.cpp"
     4          {Function} "bar" -> "int"
                  {Block}
     5              {Variable} "i" -> "int"
     4            {Parameter} "A" -> "int *"
     4            {Parameter} "X" -> "int"
```

--show-block-attributes

Show (or hide) the block attributes, which describe a typedef or template alias.

--show-class

Show (or hide) the class objects, which describe a C++ class type.

Example: Show only class objects

```
$ diva example_09.o --show-none --show-class

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     4          {Class} "A"
```

Example: Hiding DIVA class objects

```
$ diva example_09.o --no-show-class

           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"

  {Source} "example_09.cpp"
     2          {Alias} "CHAR" -> "char"
     5            {Member} private "a" -> "int"
     8          {Variable} "a" -> "A"
    10          {Function} "foo" -> "CHAR"
    10            {Parameter} "p" -> "char *"
    12            {Variable} "c" -> "CHAR"
```

--show-enum

Show (or hide) the enum objects, which describe a declaration of an enumeration type and its set of named constants (enumerator list).

--show-function

Show (or hide) the function objects, which describe a function or subroutine. A function may be defined with inline in the C/C++, but the attribute “inlined” is only given if the compiler actually inlined it. When * is given for the function name, then it is a function pointer.

Example: Showing the only the function objects.

```
$ diva example_12.o --show-none --show-function

           {InputFile} "example_12.o"

             {CompileUnit} "example_12.cpp"

  {Source} "example_12.cpp"
     4             {Function} "A::B::foo" -> "void"
                       - Is declaration
    10             {Function} "A::C::bar" -> "void"
                       - Is declaration
     4         {Function} "foo" -> "void"
                   - Declaration @ example_12.cpp,4
    10         {Function} "bar" -> "void"
                   - Declaration @ example_12.cpp,10
```

--show-member

Show (or hide) the member objects, which describe a class member.

Example: Showing only the member objects.

```
$ diva example_04.o --show-none --show-member

          {InputFile} "example_04.o"

            {CompileUnit} "example_04.cpp"

  {Source} "example_04.cpp"
     6           {Member} private "x" -> "int"
     6           {Member} private "y" -> "int"

  {Source} "ios_base.h"
   547              {Member} private "_S_refcount" -> "_Atomic_word"
   548              {Member} private "_S_synced_with_stdio" -> "bool"
```

--show-namespace

Show (or hide) the namespace objects, which describe a namespace. Anonomous namespaces will not print anything after {Namespace}

Example: Showing only the namespace objects.

```
$ diva example_04.o --show-none --show-namespace

           {InputFile} "example_04.o"

             {CompileUnit} "example_04.cpp"

  {Source} "debug.h"
    54          {Namespace} "__gnu_debug"

  {Source} "c++config.h"
   184          {Namespace} "std"

  {Source} "debug.h"
    48           {Namespace} "std::__debug"
```

--show-parameter

Show (or hide) the parameter objects, which describe a formal parameter of a parameter list for a function. "…" is a special parameter name that reflects an unspecified number of parameters.

Example: Showing only the function parameter objects.

```
$ diva example_13.o --show-none --show-parameter

           {InputFile} "example_13.o"

             {CompileUnit} "example_13.cpp"

  {Source} "example_13.cpp"
     4           {Parameter} "A" -> "int *"
     4           {Parameter} "X" -> "int"
```

--show-primitivetype

Show (or hide) the primitive type objects, which describe a language primitive used in the program and the number of bytes used.

Example: Showing only the primitive type objects.

```
$ diva example_13.o --show-none --show-primitivetype

           {InputFile} "example_13.o"

             {CompileUnit} "example_13.cpp"
               {PrimitiveType} -> "int"
                   - 4 bytes
```

--show-struct

Show (or hide) the struct objects, which describe a structure type.

Example: Showing only the struct type objects.

```
$ diva example_14.o --show-none --show-struct
           {InputFile} "example_14.o"

             {CompileUnit} "example_14.cpp"

  {Source} "example_14.cpp"
     2         {Struct} "bar"
     6           {Struct} "bar"
     9             {Struct} "bar"
```

--show-template

Show (or hide) the template objects, which describe a C++ template class or template function.

--show-union

Show (or hide) the union objects, which describe a union type.

--show-using

Show (or hide) the using objects, which describe an imported namespace, a namespaced function or namespaced variable.

Example: Showing only the using objects.

```
$ diva example_04.o --show-none --show-using
           {InputFile} "example_04.o"

             {CompileUnit} "example_04.cpp"

  {Source} "example_04.cpp"
     3         {Using} namespace "std"

  {Source} "debug.h"
    56           {Using} namespace "__debug"
```

--show-variable

Show (or hide) the variable objects, which describe a declaration for a variable.

Example: Showing only the variable objects.

```
$ diva example_04.o --show-none --show-variable

          {InputFile} "example_04.o"

            {CompileUnit} "example_04.cpp"

  {Source} "example_04.cpp"
    15           {Variable} "p1" -> "Point"
    16           {Variable} "p2" -> "Point"

  {Source} "iostream"
    74         {Variable} "std::__ioinit" -> "std::ios_base::Init"
    74           {Variable} "__ioinit" -> " std::ios_base::Init"
```

4 DIVA advanced options

4.1 Printing DWARF offsets and tag information

DWARF experts may wish to see DWARF information within the logical scope. There is some (limited) capability to output the DWARF information. All DWARF specific information will be printed inside [] braces.

```
$ diva example_01.o  --show-DWARF-offset --show-DWARF-parent

                                  {InputFile} "example_01.o"

[0x0000000b][0x00000000]            {CompileUnit} "example_01.cpp"

                           {Source} "example_01.cpp"
[0x00000026][0x0000000b]     2         {Function} "foo" -> [0x00000026]"void"
[0x00000043][0x00000026]     2           {Parameter} "c" -> [0x00000060]"char"
[0x00000051][0x00000026]     4           {Variable} "i" -> [0x00000067]"int"
```

Figure 11 DIVA output with DWARF offsets

```
$ diva example_01.o  --show-DWARF-tag

                                   {InputFile} "example_01.o"

[DW_TAG_compile_unit]                 {CompileUnit} "example_01.cpp"

                            {Source} "example_01.cpp"
[DW_TAG_subprogram]            2         {Function} "foo" -> "void"
[DW_TAG_formal_parameter]      2           {Parameter} "c" -> "char"
[DW_TAG_variable]              4           {Variable} "i" -> "int"
```

Figure 12 DIVA output with DWARF tags

Using the DWARF offsets, the DIVA objects can be linked with other DWARF tools, such as DwarfDump. It shows the offset within the *.debug_info* section where the DWARF debugging information for this object is stored. It also shows the source code files associated with the executable instructions. The offset within the *.debug_info* section is referred to as the Debug Information Entry (DIE). This DIE offset can be used to cross-reference items within the DIVA output or cross-reference the DWARF into other tools such DwarfDump.

Figure 13. DWARF tree showing the offsets and DIEs

4.2 Sorting based on the DWARF offset

Example: Sort the DIVA output by DWARF offset - the --show-offset is also added to show the DWARF attributes in [] brackets.

```
$ diva example_09.o --show-dwarf-offset --sort=offset

                        {InputFile} "example_09.o"

[0x0000000b]              {CompileUnit} "example_09.cpp"

               {Source} "example_09.cpp"
[0x00000026]     8          {Variable} "a" -> [0x0000003b]"A"
[0x0000003b]     4          {Class} "A"
[0x00000043]     5            {Member} private "a" -> [0x00000051]"int"
[0x00000058]    10          {Function} "foo" -> [0x00000058]"CHAR"
[0x00000079]    10            {Parameter} "p" -> [0x000000a8]"char *"
[0x00000087]    12            {Variable} "c" -> [0x00000096]"CHAR"
[0x00000096]     2          {Alias} "CHAR" -> [0x000000a1]"char"
```

4.3 Advanced command line options

--show-codeline

The --show-codeline will print the objects that have a source code line number associated with it [.debug_line]. If no other object is associated with that source code line, then {CodeLine} will be printed. The output will also include the parent block objects to which the line belongs.

Example: Showing all DIVA line objects

```
$ diva example_13.o --show-none --show-codeline

     {InputFile} "example_13.o"

             {CompileUnit} "example_13.cpp"

  {Source} "example_13.cpp"
     4          {CodeLine}
     5          {CodeLine}
     6          {CodeLine}
     5          {CodeLine}
     9          {CodeLine}
     9          {CodeLine}
```

--show-combined

The --show-combined option combines all the segmented debug information entries for an object. There are two well defined cases: functions and namespaces.

For namespaces, if the C++ source code contains several definitions of the same namespace, some compilers generate a single debugging information entries that includes all of its definitions, whilst other compilers will generate several debugging information entries containing the individual definitions [DW_AT_extension]. Using --show-combined, the DIVA output will show only one reference to a namespace and all its definitions.

For functions, when a member function is defined outside of the class, most compilers generate two debugging information entries for the declaration and definition [DW_AT_specification]. Using --show-combined, the DIVA output will have only one entry for the function under the class definition.

Examples: Using the following sample code:

```
 1 /* Example_12.cpp */
 2 namespace A {
 3   namespace B {
 4     void foo() {};
 5   }
 6 }
 7
 8 namespace A {
 9   namespace C {
10     void bar() {};
11   }
12 }
```

Example: There are two instances of the namespace A

```
$ diva example_12.o

           {InputFile} "example_12.o"

             {CompileUnit} "example_12.cpp"

  {Source} "example_12.cpp"
     2          {Namespace} "A"
     3            {Namespace} "A::B"
     4              {Function} "A::B::foo" -> "void"
                        - Is declaration
     9            {Namespace} "A::C"
    10              {Function} "A::C::bar" -> "void"
                        - Is declaration
     4          {Function} "foo" -> "void"
                    - Declaration @ example_12.cpp,4
    10          {Function} "bar" -> "void"
                    - Declaration @ example_12.cpp,10
```

Example: All objects belonging to the namespace A are represented together

```
$ diva example_12.o --show-combined

          {InputFile} "example_12.o"

            {CompileUnit} "example_12.cpp"

  {Source} "example_12.cpp"
     2          {Namespace} "A"
     3            {Namespace} "A::B"
     4              {Function} "A::B::foo" -> "void"
                        - Is declaration
     9            {Namespace} "A::C"
    10              {Function} "A::C::bar" -> "void"
                        - Is declaration
     4           {Function} "foo" -> "void"
                     - Declaration @ example_12.cpp,4
    10           {Function} "bar" -> "void"
                     - Declaration @ example_12.cpp,10
```

Examples: Using the following sample code:

```
 1 /* Example_15.cpp */
 2 class bar {
 3   public:
 4     void foo();
 5 };
 6
 7 void bar::foo()
 8 {
 9 }
10
11 bar b;
```

Example: The second reference to function foo does not belong to class bar

```
$ diva example_15.o

          {InputFile} "example_15.o"

            {CompileUnit} "example_15.cpp"

  {Source} "example_15.cpp"
     2          {Class} "bar"
     4            {Function} "bar::foo" -> "void"
                      - Is declaration
                    {Parameter} -> "bar *"
     4          {Function} "foo" -> "void"
                    - Declaration @ example_15.cpp,4
                  {Parameter} "this" -> "bar *"
    11          {Variable} "b" -> "bar"
```

Example: Both reference to function foo belong to the class bar

```
$ diva example_15.o --show-combined

          {InputFile} "example_15.o"

            {CompileUnit} "example_15.cpp"

  {Source} "example_15.cpp"
     2          {Class} "bar"
     4            {Function} "bar::foo" -> "void"
                      - Is declaration
                    {Parameter} -> "bar *"
     4          {Function} "foo" -> "void"
                    - Declaration @ example_15.cpp,4
                  {Parameter} "this" -> "bar *"
    11          {Variable} "b" -> "bar"
```

--show-dwarf-offset

The --show-dwarf-offset option prints the DWARF offset associated with the DIVA object. These offsets are consistent with the DWARF offsets shown in other tools, such as DwarfDump.

Example:

```
$ diva example_01.o --show-DWARF-offset

                        {InputFile} "example_01.o"

[0x0000000b]              {CompileUnit} "example_01.cpp"

               {Source} "example_01.cpp"
[0x00000026]     2          {Function} "foo" -> [0x00000026]"void"
[0x00000043]     2            {Parameter} "c" -> [0x00000060]"char"
[0x00000051]     4            {Variable} "i" -> [0x00000067]"int"
```

--show-dwarf-parent

The --show-dwarf-parent option prints the DWARF offset for the parent to each DIVA object parent (with respect to the lexical scopes).

Example:

```
$ diva example_01.o --show-DWARF-offset --show-DWARF-parent

                                    {InputFile} "example_01.o"

[0x0000000b][0x00000000]              {CompileUnit} "example_01.cpp"

                             {Source} "example_01.cpp"
[0x00000026][0x0000000b]     2          {Function} "foo" ->
[0x00000026]"void"
[0x00000043][0x00000026]     2            {Parameter} "c" ->
[0x00000060]"char"
[0x00000051][0x00000026]     4            {Variable} "i" ->
[0x00000067]"int"
```

--show-dwarf-tag

The --show-dwarf-tag option prints the DWARF Tag associated to the DIVA object.

Example:

```
$ diva example_01.o --show-DWARF-tag

                                     {InputFile} "example_01.o"

[DW_TAG_compile_unit]                   {CompileUnit} "example_01.cpp"

                              {Source} "example_01.cpp"
[DW_TAG_subprogram]               2          {Function} "foo" -> "void"
                                                 - Lines 3:5
[DW_TAG_formal_parameter]         2            {Parameter} "c" -> "char"
[DW_TAG_variable]                 4            {Variable} "i" -> "int"
```

--show-generated

The --show-generated option includes any objects generated by the compiler that do not have a direct reference in the source code. As such these objects do not have an associated line number or filename. (There can be false positives when comparing two DIVA outputs using this option because this type of debug information can be represented in many ways.)

Examples: Using the following sample code:

```
 1 /* Example_04.cpp */
 2 #include <iostream>
 3 using namespace std;
 4
 5 class Point {
 6   int x, y;
 7 public:
 8   Point(int i, int j) { x = i; y = j; }
 9   int getX() { return x; }
10   int getY() { return y; }
11 };
12
13 void foo()
14 {
15   Point p1(10, 20);
16   Point p2 = p1;
17   cout << "x = " << p2.getX() << " y = " << p2.getY();
18 }
```

Example: Showing the compiler generated objects not represented in the code

```
$ diva example_04.o --show-generated
```

--show-global

The --show-global option prints an 'X' character at the start of the lines where the DIVA object is a global object.

--show-golden

The --show-golden option prints a 'G' character at the start of the lines where the DIVA object has been marked as golden. All DIVA objects are initially marked as golden, individual DIVA objects can subsequently be marked as not-golden so that it will be ignored during any future DIVA comparisons.

Example: ** This is not yet implemented - the output is a mock-up

```
$ diva example_01.o --show-golden

G     {InputFile} 'example_01.o'
G       {CompileUnit} 'example_01.cpp'

G{Source} 'example_01.cpp'
G  2        {Function} 'foo' -> 'void'
G  2          {Parameter} 'c' -> 'char'
G  4          {Variable} 'i' -> 'int'
G  3          {CodeLine}
G  4          {CodeLine}
G  5          {CodeLine}
G  5          {CodeLine}
```

--show-indent

The --show-indent option adds the indentation associated with the object scope. As a visual indication to show the parent relationship, the indentation adds 2 spaces for each level object's lexical scope depth,

Example: Printed with no indents for child objects

```
$ diva example_01.o --show-indent

           {InputFile} "example_01.o"

           {CompileUnit} "example_01.cpp"

  {Source} "example_01.cpp"
     2         {Function} "foo" -> "void"
     2           {Parameter} "c" -> "char"
     4           {Variable} "i" -> "int"
```

Example: Printed with no indents for child objects

```
$ diva example_01.o --no-show-indent

           {InputFile} "example_01.o"

           {CompileUnit} "example_01.cpp"

  {Source} "example_01.cpp"
     2     {Function} "foo" -> "void"
     2     {Parameter} "c" -> "char"
     4     {Variable} "i" -> "int"
```

--show-level

The --show-level option prints the DIVA object lexical level. The level for the input file is -1 and a compilation unit is zero.

Example: Printed with no indents but with the indent level as a digit

```
$ diva example_01.o --no-show-indent --show-level

              {InputFile} "example_01.o"

              {CompileUnit} "example_01.cpp"

001  {Source} "example_01.cpp"
001     2     {Function} "foo" -> "void"
002     2     {Parameter} "c" -> "char"
002     4     {Variable} "i" -> "int"
```

--show-only-globals

The --show-only-globals option filters the DIVA output to only show the global objects. These are the DIVA objects that are referenced across multiple {CompileUnit}s.

Example: Using the following sample code:

```
$ clang++ -w -g -O0 example_16_local.cpp example_16_global.cpp
example_16.cpp -o example_16_lto.elf -flto
```

The class 'Global' will be referenced across modules.

```
 1 /* Example_16_global.h */
 2 class Global {
 3   public:
 4     int foo(int g);
 5   private:
 6     int m_g;
 7 };
```

This module uses only the 'Global' class.

```
 1 /* Example_16_global.cpp */
 2 #include "example_16_global.h"
 3
 4 int Global::foo(int g)
 5 {
 6   m_g = g * 620;
 7   return m_g;
 8 }
```

The class 'local' will be referenced in only one module.

```
 1 /* Example_16_local.h */
 2 class Local {
 3   public:
 4     int foo(int l);
 5   private:
 6     int m_l;
 7 };
 8
 9 int do_local(int l);
10 int do_global(int g);
```

This module uses both 'Global' and 'Local' classes.

```
 1 /* Example_16_local.cpp */
 2 #include "example_16_local.h"
 3 #include "example_16_global.h"
 4
 5 int Local::foo(int l)
 6 {
 7   m_l = l * 505;
 8  return m_l;
 9 }
10
11 __attribute__((optnone))
12 int do_local(int l)
13 {
14   Local local;
15   return local.foo(l);
16 }
17
18 __attribute__((optnone))
19 int do_global(int g)
20 {
21   Global global;
22   return global.foo(g);
23 }
```

This is the main module.

```
 1 /* Example_16.cpp */
 2 #include "example_16_local.h"
 3 #include "example_16_global.h"
 4
 5 int main()
 6 {
 7   int l = 505;
 8   int g = 620;
 9   do_local(l);
10   do_global(g);
11 }
```

Example: The DIVA output only showing the globals

```
$ diva example_16_lto.elf --show-only-globals
```

--show-only-locals

The --show-only-locals options filters the DIVA output to only show the local objects. These are the DIVA objects that are only referenced within its own CU.

Example: The DIVA output only showing the locals

```
$ diva example_16_lto.elf --show-only-locals
```

--show-qualified

The --show-qualified option prints the DIVA types with the qualified parent name hierarchy information.

Example: Using the following sample code:

```
 1 /* Example_14.cpp */
 2 struct bar {};
 3 typedef char CHAR;
 4
 5 namespace nsp_1 {
 6 struct bar {};
 7   typedef char CHAR;
 8 namespace nsp_2 {
 9 struct bar {};
10     typedef char CHAR;
11   }
12 }
13
14 template<class _Ty>
15 class foo {
16 _Ty b;
17 };
18
19 CHAR a;
20 nsp_1::CHAR b;
21 nsp_1::nsp_2::CHAR c;
22
23 foo<bar> b1;
24 foo<nsp_1::bar> b2;
25 foo<nsp_1::nsp_2::bar> b3;
```

Example: The DIVA output with a qualified string for the 'CHAR' and 'bar' types, showing the associated parents

Example: Showing only the template objects

```
$ diva example_14.o --show-none --show-template

         {InputFile} "example_14.o"

           {CompileUnit} "example_14.cpp"

{Source} "example_14.cpp"
 15         {Class} "foo<bar>"
                - Template
              {TemplateParameter} "foo<nsp_1::nsp_2::bar>:: _Ty" <- "bar"
 15         {Class} "foo<nsp_1::bar>"
                - Template
              {TemplateParameter} "foo<nsp_1::bar>:: _Ty" <- "bar"
 15         {Class} "foo<nsp_1::nsp_2::bar>"
                - Template
              {TemplateParameter} "foo<nsp_1::nsp_2::bar>:: _Ty" <-
                                                    "nsp_1::nsp_2::bar"
```

--show-ranges

With the --show-ranges option, the DIVA output will include the range objects [.debug_ranges] that represent a section with executable machine code.

Example: Showing all DIVA range objects

```
$ diva example_09.o --show-none --show-ranges

* Ranges * : {CompileUnit} 'example_09.cpp'
[0x00000000,0x00000017][001] {Function} 'foo'
           {InputFile} "example_09.o"

             {CompileUnit} "example_09.cpp"
```

The --show-underlying option prints the underlying type for objects that are symbols.

Examples: Using the following sample code:

```
 1 /* Example_06.cpp */
 2 struct S {
 3   int s;
 4 };
 5
 6 typedef int INT;
 7 typedef INT INTEGER;
 8 typedef S STRUCT;
 9
10 int var_0;
11 INT var_1;
12 INTEGER var_2;
13
14 STRUCT var_3;
```

Example: Showing the type defines used

```
$ diva example_06.o --show-underlying

           {InputFile} "example_06.o"

             {CompileUnit} "example_06.cpp"

  {Source} "example_06.cpp"
     2         {Struct} "S"
     3           {Member} public "s" -> "int"
     6         {Alias} "INT" -> "int"
     7         {Alias} "INTEGER" -> "INT"
     8         {Alias} "STRUCT" -> "S"
    10         {Variable} "var_0" -> "int"
    11         {Variable} "var_1" -> "INT"
    12         {Variable} "var_2" -> "INTEGER"
    14         {Variable} "var_3" -> "STRUCT"
```

--show-void

The --show-void option prints the 'void' type for objects that have any reference to that type. Note that not all compilers emit the 'void' type in the debug information.

Examples: Using the following sample code:

```
1 /* Example_07.cpp */
2 void *pv;
3 int *pi;
4 void foo()
5 {
6 }
```

Example:

```
$ diva example_07.o --no-show-void
           {InputFile} "example_07.o"

             {CompileUnit} "example_07.cpp"

  {Source} "example_07.cpp"
     2         {Variable} "pv" -> "*"
     3         {Variable} "pi" -> "int *"
     4         {Function} "foo" -> ""
```

Example:

```
$ diva example_07.o --show-void
           {InputFile} "example_07.o"

             {CompileUnit} "example_07.cpp"

  {Source} "example_07.cpp"
     2         {Variable} "pv" -> "void *"
     3         {Variable} "pi" -> "int *"
     4         {Function} "foo" -> "void"
```

4.4 DWARF warning options

--warning-all — The --warning-all option enables all the other --warning-* options listed in this table.

--warning-attribute — The --warning-attribute option prints the DWARF attributes (See DWARF Debugging Information Format document), that are invalid or not supported by DIVA. The list includes the invalid DWARF attribute and the DIE offsets where that attribute is used.

Example:

```
$ diva example_01.o --warning-attribute
```

--warning-locations — The --warning-locations option enables warnings for variable, parameter and member location issues.

--warning-reference — The --warning-reference option prints the DWARF tags (See DWARF Debugging Information Format document), that are invalid or not supported by DIVA. The list includes the invalid DWARF reference and the DIE offsets where that reference is used.

Example:

```
$ diva example_01.o --warning-reference
```

--warning-tag — The --warning-tag option prints the DWARF tags (See DWARF Debugging Information Format document), that are invalid or not supported by DIVA. The list includes the invalid DWARF tag and the DIE offsets where that tag is used.

Example:

```
$ diva example_01.o --warning-tag
```

5 DIVA object output format

DIVA represents the C/C++ source code in a high-level and simple abstraction. It represents the following objects: type, scope, symbol and line. These objects are the basic elements used in the C/C++ programming language. If the debug information contained within the input ELF file is valid DWARF but is structurally incorrect, such as a {CodeLine} being a lexical parent of a {Function}, DIVA will print the objects as specified.

5.1 Textual output format

WARNING - Whilst DIVA is in beta, this textual structure/format may change

The textual representation of the DIVA objects follow the following genral structure and Table 2 describes the specific structure of each DIVA object.

```
{<DIVA object>} "<name of instance>" -> "<type>"
    - <attribute details>
    - <attribute details>
```

Table 2 DIVA objects printed in textual form

DIVA object and meaning	DIVA object syntax and example	--show-brief (default)	--show-all

{Alias}
 	
An alias, which can be a typedef or template alias.
	Syntax:
	`{Alias} "<name of alias>" -> "<type>"`
	
	Example:
	`{Alias} "INTEGER" -> "int"`
	`{Alias} 'ptr<int>' -> '* int'`
	
	Command line options:
	`--show-alias`
		✓	✓

{Block}
 	
A lexical block.
	Syntax:
	`{Block}`
	`- try`
	`- catch`
	
	Examples:
	`{Block}`
	`{Block}`
	`- catch`
	
	Command line options:
	`--show-block`
	`--show-block-attributes`
		✓ Without attributes	✓ With attributes

{Class}
 	
A C++ class type
	Syntax:
	`{Class} "<name of class>"`
	`- Template`
	`- <public|private|protected> "<class inherited from>"`
	`- <public|private|protected> "<class inherited from>"`
	
	Examples:
	`{Class} "Cat"`
	`- public "legs"`
	`- protected "tail"`
	
	Command line options:
	`--show-class`
		✓	✓

{CodeLine}
 	
Line information for the code in the source file.
 	
The attributes are only shown when the option is explicitly added, as these are expected to be used only by the advanced users.
	Syntax:
	`{CodeLine}`
	`- NewStatement`
	`- PrologueEnd`
	`- EndSequence`
	`- BasicBlock`
	`- Discriminator`
	`- EpilogueBegin`
	
	Examples:
	`{CodeLine}`
	`{CodeLine}`
	`- NewStatement`
	`- Discriminator`
	
	Command line options:
	`--show-codeline`
	`--show-codeline-atrributes`
		✓ Without attributes	✓ Without attributes

{CompileUnit}	Syntax:
	`{CompileUnit} "<source code filename>"`
	
	Examples:
	`{CompileUnit} "helloworld.cpp"`
	
	Command line options:
	`(Always enabled)`
		✓	✓

The source file, which represents the text and data contributed to an executable by a single object file. It may be derived from several other source files that where included as pre-processed or “include files.”			

{Enum}
 	
Declaration of an enumeration type and its set of named constants (enumerator list).
	Syntax:
	`{Enum} <class> “<name of instance>” -> “<type>”`
	`- “<enumerator instance>”`
	`- “<enumerator instance>”`
	
	Examples:
	`{Enum} “days”`
	`- “mon”=1`
	`- “tue”=2`
	`- “wed”=3`
	`{Enum} class “months” -> “int”`
	`- “jan”=1`
	`- “feb”=2`
	`- “mar”=3`
	
	Command line options:
	`--show-enum`
		✓	✓

{InputFile}
The input file to DIVA.
	Syntax:
	`{InputFile} “<name of input file>”`
	
	Examples:
	`{InputFile} “helloworld.o”`
	
	Command line options:
	`(always enabled)`
		✓	✓

{Function}
 	
A function or subroutine.
 	
A function may be defined with inline in the C/C++, but the attribute “inlined” is only given if the compiler actually inlined it.
 	
When * is given for the function name, then it is a function pointer.
 	
The entrypoint attribute is given when there are multiple entry points into the same function (not used in C/C++).
	Syntax:
	`{Function} <static|inline> “<name of instance or pointer>” -> “<return type>”`
	`- Declaration @[<file>,]<line>`
	`- No declaration`
	`- Is declaration`
	`- Template`
	`- Inlined`
	`- Lines <min line number>:<max line number>`
	
	Examples:
	`{Function} static inline “helloworld” -> “int”`
	`- inlined`
	`- Lines 10:25`
	`- Declaration @ helloworld.h,23`
	`{Function} * -> bool`
	
	Command line options:
	`--show-function`
	`--show-function-attributes`
		✓
		Without attributes
			✓
			With attributes

{Location}
 	
A variable, parameter etc. value location for the given code line range, where the location may be on the stack, in memory, or in a register.
	Syntax:
	`{Location} “<line range min>:<line range max>”`
	`- Address: <address>`
	`- Stack: <expression>`
	`- Register: <register name>`
	`- Missing`
	
	Examples:		

When the debug information does not describe a location for a range of code lines, then "missing locaton" is given.	{Location} 9:9 - Address: 0x0080000 {Location} 10:18 - Stack: -8 {Location} 19:24 - Missing {Location} 25:26 - Register: r5 + 0 : 4 - Register: r9 + 0 : 4 {Location} 27:29 - Register: rdi + 0 Command line options: --show-location		
{Member}	Syntax: {Member} <public|private|protected> <static> "<name of instance or pointer>" -> "<type>" - Location coverage: <value>% Examples: {Member} private static "count" -> "const int" - Location coverage: 88% {Member} * -> "const int" Command line options: --show-member --show-location-coverage	✓ Without location coverage	✓ With location coverage
{Namespace} A namespace. Anonomous namespaces will not print anything after {Namespace}	Syntax: {Namespace} "<name of namespace>" Examples: {Namespace} "mynamespace" {Namespace} Command line options: --show-namespace	✓	✓
{Parameter} Formal parameter of a parameter list for a function. "…" is a special parameter name that reflects an unspecified number of parameters.	Syntax: {Parameter} "<name of instance>" -> "<type>" - Location coverage: <value>% {TemplateParameter} "<name of instance>" <- "<template argument>" {TemplateParameter} "<name of instance>" <- "<type>" <- "<type>" <- "<type>" Examples: {Parameter} "pArray" -> "const int" 5,23 - Location coverage: 58% {Parameter} "T" <- "int" Command line options: --show-parameter --show-location-coverage	✓ Without location coverage	✓ With location coverage
{PrimitiveType} Language primitive used in the program and the number of bytes used.	Syntax: {PrimitiveType} -> "<type>" - <size> bytes Examples: {PrimitiveType} -> "int" - 32 bytes {PrimitiveType} -> "long long" - 64 bytes Command line options: --show-primitivetype		✓
{Struct} A structure type.	Syntax: {Struct} "<name of structure>" - Template - <public|private|protected> "<struct inherited from>" - <public|private|protected> "<struct inherited from>" Examples:	✓	✓

	{Struct} "data" - public "object" Command line options: --show-struct		
{Union} A union type.	Syntax: {Union} "<name of union instance>" - Template Command line options: --show-union	✓	✓
{Using} An imported namespace, a namespaced function or namespaced variable	Syntax: {Using} <type|namespace|variable|function> "<name of instance>" Examples: {Using} namespace "std" {Using} namespace "STD" - "std" {Using} variable "std::a" {Using} type "std::a" {Using} function "std::foo" Command line options: --show-using	✓	✓
{Variable} Declaration for a variable.	Syntax: {Variable} <static> "<name of instance>" -> "<type>" - Location coverage: <value>% Examples: {Variable} static "text" -> "volatile char *" - Location coverage: 78% Command line options: --show-variable --show-location-coverage	✓ Without location coverage	✓ With location coverge

5.2 YAML output format

WARNING - Whilst DIVA is in beta, this YAML structure/format may change and there will be no attempt at backwards compatibility.

LIMITATION - In the currect DIVA version, there is no option to hide any DIVA object or DIVA attribute in the YAML output.

Each instance of a DIVA object will follow the form described in object: "Name of DIVA object"

```
name: "<name of instance>"
type: "<type>"
source:
    line: <line number>
    file: "<file path>"
dwarf:
    offset: <dwarf offset>
    tag: "<dwarf tag>"
attributes: {}
children: []
```

Figure 14. Each object instance in YAML will contain "object", "name", "type", "souce", "dwarf", "attributes" and "children" fields, regardless of whether that field contains any information. Each "dwarf" field has "offset" and "tag" sub-fields; each "source" field has a "file" and "line" field; if there is no name or type, then "null" is given. The "children" field may contain DIVA objects where these will be the direct decendants of the DIVA object. Refer to Table 3 for specific YAML details of each DIVA object.

```
- object: "Name of DIVA object"
  name: "<name of instance>"
  type: "<type>"
  source:
      line: <line number>
      file: "<file path>"
  dwarf:
      offset: <dwarf offset>
      tag: "<dwarf tag>"
  attributes: {}
  children: []
```

Figure 14. DIVA debug information YAML format

Table 3 DIVA objects printed in YAML form

DIVA object and meaning	**DIVA object syntax**

{Alias}
 	
An alias, which can be a typedef or template alias.
	Structure:
	
	`- object: "Alias"`
	`name: "<name of alias>"`
	`type: "<type>"`
	`source:`
	`line: <line number>`
	`file: "<file path>"`
	`dwarf:`
	`offset: <dwarf offset>`
	`tag: "<dwarf tag>"`
	`attributes: {}`
	`children:`
	`- <children of object>`

{Block}
 	
A lexical block.
	Structure:
	
	`- object: "Block"`
	`name: null`
	`type: null`
	`source:`
	`line: <line number>`
	`file: "<file path>"`
	`dwarf:`
	`offset: <dwarf offset>`
	`tag: "<dwarf tag>"`
	`attributes:`
	`try: <bool>`
	`catch: <bool>`
	`children:`
	`- <children of Block>`

{Class}
 	
A C++ class type
	Structure:
	
	`- object: "Class"`
	`name: "<name of class>"`
	`type: null`
	`source:`
	`line: <line number>`
	`file: "<file path>"`
	`dwarf:`
	`offset: <dwarf offset>`
	`tag: "<dwarf tag>"`
	`attributes:`
	`is_template: <bool>`
	`inherits_from:`
	`- parent: "<parent <class|struct> name>"`
	`access_specifier: "<public|private|protected>"`
	`- parent: "<parent <class|struct> name>"`
	`access_specifier: "<public|private|protected>"`
	`children:`
	`- <children of Class>`

{CodeLine}	Structure:

Line information for the code in the source file.
 	
The attributes are only shown when the option is explicitly added, as these are expected to be used only by the advanced users.
	<pre>- object: "CodeLine"
	  name: null
	  type: null
	  source:
	  line: <line number>
	  file: "<file path>"
	  dwarf:
	  offset: <dwarf offset>
	  tag: null
	  attributes:
	  NewStatement: <bool>
	  PrologueEnd: <bool>
	  EndSequence: <bool>
	  BasicBlock: <bool>
	  Discriminator: <bool>
	  EplilogueBegin: <bool>
	  children: []</pre>

{CompileUnit}
 	
The source file, which represents the text and data contributed to an executable by a single object file. It may be derived from several other source files that where included as pre-processed or "include files."
	Structure:
	
	<pre>- object: "CompileUnit"
	  name: "<name of source code file>"
	  type: null
	  source:
	  line: null
	  file: null
	  dwarf:
	  offset: <dwarf offset>
	  tag: "<dwarf tag>"
	  attributes: {}
	  children:
	  - <children of CompileUnit></pre>

{Enum}
 	
Declaration of an enumeration type and its set of named constants (enumerator list).
	Structure:
	
	<pre>- object: "Enum"
	  name: "<name of enum>"
	  type: "<enum type>"
	  source:
	  line: <line number>
	  file: "<file path>"
	  dwarf:
	  offset: <dwarf offset>
	  tag: "<dwarf tag>"
	  attributes:
	  class: true
	  enumerators:
	  - enumerator: "<enumerator name>"
	  value: <value>
	  - enumerator: "<enumerator name>"
	  value: <value>
	  children: []</pre>

{InputFile}
The input file to DIVA.
	Structure:
	
	<pre>- input_file: "<input file name>"
	  output_version: <diva YAML ouput version number>
	  objects:
	  - <YAML output tree></pre>

{Function}	Structure:

A function or subroutine.
 	
A function may be defined with inline in the C/C++, but the attribute “inlined” is only given if the compiler actually inlined it.
 	
When * is given for the function name, then it is a function pointer.
 	
The entrypoint attribute is given when there are multiple entry points into the same function (not used in C/C++).
	<pre>- object: “Function”
	name: “<name of function instance>”
	type: “<return type>”
	source:
	line: <line number>
	file: “<file path>”
	dwarf:
	offset: <dwarf offset>
	tag: “<dwarf tag>”
	attributes:
	declaration:
	file: “<file name>”
	line: <line number>
	is_template: <bool>
	static: <bool>
	inline: <bool>
	is_inlined: <bool>
	is_declaration: <bool>
	children:
	- <children of function></pre>

{Location}
 	
A variable, parameter etc. value location for the given code line range, where the location may be on the stack, in memory, or in a register.
 	
When the debug information does not describe a location for a range of code lines, then “missing locaton” is given.
	Structure:
	
	<pre>- object: “Location”
	name: null
	type: null
	source:
	line: <line number>
	file: “<file path>”
	dwarf:
	offset: <dwarf offset>
	tag: “<dwarf tag>”
	attributes:
	line_range:
	lower_bound: “<line range min>”
	upper_bound: “<line range max>”
	address: <address>
	stack: <expression>
	registers:
	- “<register name>”
	missing: <bool>
	children: []</pre>

{Member}	Structure:
	
	<pre>- object: “Member”
	name: “<name of instance if not pointer>”
	type: “<type>”
	source:
	line: <line number>
	file: “<file path>”
	dwarf:
	offset: <dwarf offset>
	tag: “<dwarf tag>”
	attributes:
	access_specifier: “<public|private|protected>”
	static: <bool>
	children:
	- <children of member></pre>

{Namespace}
 	
A namespace. Anonomous namespaces will not print anything after {Namespace}
	Structure:
	
	<pre>- object: “Namespace”
	name: “<name of namespace>”
	type: null
	source:
	line: <line number>
	file: “<file path>”
	dwarf:
	offset: <dwarf offset>
	tag: “<dwarf tag>”
	attributes: {}
	children:
	- <children of namespace></pre>

{Parameter}	Structure:

Formal parameter of a parameter list for a function.
 	
"…" is a special parameter name that reflects an unspecified number of parameters.
	<pre>- object: "Parameter"
	name: "<name of instance>"
	type: "<type"
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes: {}
	children: []</pre>

{PrimitiveType}
 	
Language primitive used in the program and the number of bytes used.
	Structure:
	<pre>- object: "PrimitiveType"
	name: "<name of primitive type>"
	type: "<type>"
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	size: <size in bytes>
	children: []</pre>

{Struct}
 	
A structure type.
	Structure:
	<pre>- object: "Struct"
	name: "<name of structure>"
	type: null
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	is_template: <bool>
	inherits_from:
	- parent: "<parent <class|struct> name>"
	access_specifier: "<public|private|protected>"
	- parent: "<parent <class|struct> name>"
	access_specifier: "<pulic|private|protected>"
	children:
	- <children of structure></pre>

{TemplateParameter}
 	
A C++ template class or template function.
	Structure:
	<pre>- object: "TemplateParameter"
	name: "<name of template parameter instance">">
	type: null
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	types:
	- "<type>"
	children: []</pre>

{Union} A union type.	Structure:

	<pre>- object: "Union"
	name: "<name of union instance>"
	type: null
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	is_template: <bool>
	children:
	- <children of object></pre>

{Using}
 	
An imported namespace, a namespaced function or namespaced variable
	Structure:
	
	<pre>- object: "Using"
	name: "<name of namespace">"
	type: null
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	using_type: "<type|namespace|variable|function>"
	children: []</pre>

{Variable}
 	
Declaration for a variable.
	Structure:
	
	<pre>- object: "Variable"
	name: "<name of variable instance>"
	type: "<type>"
	source:
	line: <line number>
	file: "<file path>"
	dwarf:
	offset: <dwarf offset>
	tag: "<dwarf tag>"
	attributes:
	is_static: true
	children: []</pre>

6 Appendix

6.1 **Roadmap**

DIVA has a backlog of features to add. The major items are listed below.

6.1.1 Future releases

- Support DWARF 5 input - for verification of the new DWARF 5 standard
- Open sourcing the DIVA code base
- Support for the "Golden" attribute - this is dependant upon user-feedback. Please let us know whether you think the feature, as described in this user guide, would be useful for you.

6.2 **Known issues**

- The current DIVA implementation does not have the ability to mark individual DIVA objects as *Golden*. A user interface is required and is on the backlog of features.
- For some specific namespace combinations, the DIVA output serialization fails, and a stack overflow error is seen.
- Any DIVA block object with no associated line information will be ignored by the built-in comparison module. A workaround is to use an external diff program if a more precise information is required.

6.3 Error messages

The tool produces error messages upon encountering issues that prevent it from running further.

ERR_OPTIONS_INVALID_SEPARATOR	"Invalid or missing separator '%s'." The given separator is missing or invalid.
ERR_OPTIONS_INVALID_OPTION	"Unrecognized option '%s'." The specified argument is invalid for the given option.
ERR_OPTIONS_INVALID_ARG	"Unrecognized argument '%c' for option '-%c'." The specified argument is invalid for the given option.
ERR_OPTIONS_PARSING_EXCEPTION	"Exception raised while parsing program arguments." This error is generated when a runtime exception is raised while parsing a program argument.
ERR_OPTIONS_INVALID_BOOLEAN_ARG	"Invalid Boolean argument '%s'." This error is generated when an argument is giving a value that does not represent a logical value.
ERR_OPTIONS_INVALID_INDEX_ARG	"Invalid option '%s' for Index operation." The given option is invalid for Save/Load Index file.
ERR_OPTIONS_INVALID_REGEX	"Invalid Regular Expression '%s'." The given regular expression is invalid.
ERR_OPTIONS_INVALID_TABLE_INDEX	"Invalid option '%d' for Long Names Table." The given option has an invalid internal index.
ERR_SPLIT_UNABLE_TO_OPEN_FILE	"Unable to open file '%s' for DIVA view Split." Unable to open the given filename, while doing DIVA output Split.
ERR_INTERNAL_ERROR	"Internal Error: '%s'." An internal error has occurred. DIVA will terminate.

6.4 Glossary of terms

This table summarizes some of the most common terms used in these manual.

Term	**Meaning**
CLANG	Open source C language compiler
DIE	Debugging Information Entry
DIVA	Debug Information Visual Analyzer
DWARF	Debugging With Objectd Record Formats
ELF	Executable and Linkable Format
GCC	GNU C language Compiler
LTO	Link Time Optimization
LLVM	Open source code base used for compiler technology
ELF OBJECT	Compiler output file that uses the ELF format