ELFIO

Tutorial and User Manual

Table of Contents

[2 Introduction 2](#_Toc341017058)

[3 Getting Started With ELFIO 2](#_Toc341017059)

[3.1 ELF File Reader 2](#_Toc341017060)

[3.2 ELF Section Data Accessors 5](#_Toc341017061)

[3.3 ELFDump Utility 5](#_Toc341017062)

[4 ELFIO Library Classes 6](#_Toc341017063)

[4.1 elfio 6](#_Toc341017064)

[4.1.1 Data members 6](#_Toc341017065)

[4.1.2 Member functions 6](#_Toc341017066)

[4.2 section 6](#_Toc341017067)

[4.2.1 Member functions 6](#_Toc341017068)

[4.3 segment 6](#_Toc341017069)

[4.3.1 Member functions 6](#_Toc341017070)

# Introduction

ELFIO is a C++ library for reading and generating files in ELF binary format. This library is independent and does not require any other product. It is also cross-platform - the library uses standard ANSI C++ constructions and runs on wide variety of architectures.

While the library's implementation does make your work much easier: basic knowledge of the ELF binary format is required. Information about ELF format can be found widely on the web.

# Getting Started With ELFIO

## ELF File Reader

The ELFIO library is a header only library. No preparatory compilation steps are required. To make your application be aware about the ELFIO classes and types declarations, just include <elfio.hpp> header file. All ELFIO library declarations reside in ELFIO namespace. So, this tutorial code starts with the code:

#include <iostream>

#include <elfio.hpp> 

using namespace ELFIO 

int main( int argc, char\*\* argv )

{

if ( argc != 2 ) {

std::cout << "Usage: tutorial <elf\_file>" << std::endl;

return 1;

}

 - Include elfio.hpp header file

* - The ELFIO namespace usage

This chapter will explain how to work with the reader portion of the ELFIO library. The first step would be creation of the elfio class instance. The elfio constructor does not receive any parameters. After creation of a class object, we initialize the instance by invoking load function passing ELF file name as a parameter.

// Create elfio reader

elfio reader; 

// Load ELF data

if ( !reader.load( argv[1] ) ) { 

std::cout << "Can't find or process ELF file " << argv[1] << std::endl;

return 2;

}

 - Create elfio class instance

- Initialize the instance by loading ELF file. The function load returns ‘true’ if the ELF file was found and processed successfully. It returns ‘false’ otherwise



ELF header properties are available now. This makes it possible to request file parameters such as encoding, machine type, entry point, etc. To get the class and the encoding of the file use:

// Print ELF file properties

std::cout << "ELF file class : ";

if ( reader.get\_class() == ELFCLASS32 ) 

std::cout << "ELF32" << std::endl;

else

std::cout << "ELF64" << std::endl;

std::cout << "ELF file encoding : ";

if ( reader.get\_encoding() == ELFDATA2LSB ) 

std::cout << "Little endian" << std::endl;

else

std::cout << "Big endian" << std::endl;

 - Member function get\_class()returns ELF file class. Possible values are: ELFCLASS32 or ELFCLASS64

 - Member function get\_encoding() returns ELF file format encoding. Possible values are: ELFDATA2LSB or ELFDATA2MSB standing for little- and big-endianess correspondingly

* **Note:**

Standard ELF types, flags and constants are defined in the elf\_types.hpp header file. This file is included automatically into the project. For example: ELFCLASS32, ELFCLASS64 constants define a value for 32/64 bit architectures. Constants ELFDATA2LSB and ELFDATA2MSB define value for little- and big-endian encoding.

ELF binary files may consist of several sections. Each section has it's own responsibility: some contain executable code; others describe program dependencies; others symbol tables and so on. See ELF binary format documentation for a full description of each section.

The following code demonstrates how to find out the amount of sections the ELF file contains. The code also presents how to access particular section properties like names and sizes:

// Print ELF file sections info

Elf\_Half sec\_num = reader.sections.size(); 

std::cout << "Number of sections: " << sec\_num << std::endl;

for ( int i = 0; i < sec\_num; ++i ) {

const section\* psec = reader.sections[i]; 

std::cout << " [" << i << "] "

<< psec->get\_name() 3.gif

<< "\t"

<< psec->get\_size() 3.gif

<< std::endl;

// Access section's data

const char\* p = reader.sections[i]->get\_data(); 3.gif

}

 - Retrieve number of sections

 - Use operator[] to access a section by its number

3.png - get\_name(), get\_size() and get\_data() are member of ‘section’ class

‘sections’ member of ‘reader’ object permits obtaining the number of sections inside given ELF file. It also serves for getting access to individual section by using operator[], which returns a pointer to corresponding section's interface.

Similarly, segments of the ELF file can be processed:

// Print ELF file segments info

Elf\_Half seg\_num = reader.segments.size(); 

std::cout << "Number of segments: " << seg\_num << std::endl;

for ( int i = 0; i < seg\_num; ++i ) {

const segment\* pseg = reader.segments[i]; 

std::cout << " [" << i << "] 0x" << std::hex

<< pseg->get\_flags() 3.gif

<< "\t0x"

<< pseg->get\_virtual\_address() 3.gif

<< "\t0x"

<< pseg->get\_file\_size() 3.gif

<< "\t0x"

<< pseg->get\_memory\_size() 3.gif

<< std::endl;

// Access segments's data

const char\* p = reader.segments[i]->get\_data();3.gif

}

 - Retrieve number of segments

 - Use operator[] to access a segment by its number

3.png - get\_flags(), get\_virtual\_address(), get\_file\_size(), get\_memory\_size() and get\_data() are member methods of ‘segment’ class

In this case, segments' attributes and data are obtained by using ‘segments’ member of the ‘reader’ class.

The full text of this example comes together with ELFIO library distribution.

## ELF Section Data Accessors

To simplify creation and interpretation of the ELF sections' data, the ELFIO library comes with auxiliary classes - accessors. To the moment of this document writing, the following accessors are available:

* string\_section\_accessor
* symbol\_section\_accessor
* relocation\_section\_accessor
* note\_section\_accessor
* dynamic\_section\_accessor

More accessors may be implemented in future versions of the library.

Definitely, it is possible to extend the library by implementing additional accessors for less generic and customized purposes.

Let's see how the accessors can be used in combination with the previous ELF file reader example. For this purpose, we print out all symbols in symbol section:

if ( psec->get\_type() == SHT\_SYMTAB ) {

const symbol\_section\_accessor symbols( reader, psec );

for ( unsigned int j = 0; j < symbols.get\_symbols\_num(); ++j ) {

std::string name;

Elf64\_Addr value;

Elf\_Xword size;

unsigned char bind;

unsigned char type;

Elf\_Half section\_index;

unsigned char other;

symbols.get\_symbol( j, name, value, size, bind,

type, section\_index, other );

std::cout << j << " " << name << std::endl;

}

}

We have created ‘symbol\_section\_accessor’ instance first. Usually, accessors receive the elfio and ‘section’ objects as parameters for their constructors. get\_symbol() method is used for retrieving a particular entry in the symbol table.

## ELFDump Utility

The source code for the ELF Dumping Utility can be found in the "examples" directory; there also located more examples on how to use different ELFIO reader interfaces.

# ELFIO Library Classes

## elfio

### Data members

The ELFIO library's main class is ‘elfio’. The class contains the following public data members:

### Member functions

## section

Class ‘section’ has no public data members.

### Member functions

## segment

Class ‘segment’ has no public data members.

### Member functions