ELFIO

Tutorial and User Manual

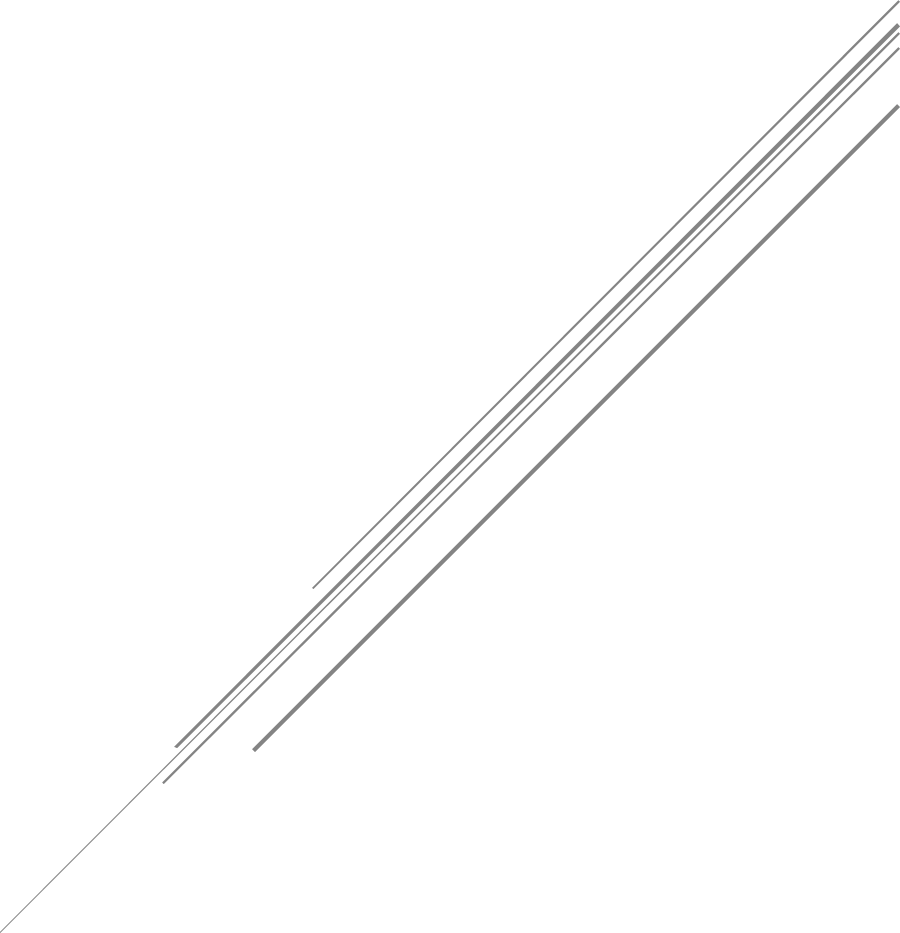


Table of Contents

[2 Introduction 2](#_Toc341613018)

[3 Getting Started With ELFIO 2](#_Toc341613019)

[3.1 ELF File Reader 2](#_Toc341613020)

[3.2 ELF Section Data Accessors 5](#_Toc341613021)

[3.3 ELFDump Utility 6](#_Toc341613022)

[3.4 ELF File Writer 6](#_Toc341613023)

[4 ELFIO Library Classes 8](#_Toc341613024)

[4.1 elfio 8](#_Toc341613025)

[4.2 section 11](#_Toc341613026)

[4.3 segment 11](#_Toc341613027)

# Introduction

ELFIO is a C++ header only library for reading and generating files in ELF binary format. It is a standalone library; it is not dependant on any other product or project. It is also a cross-platform library – written on standard ISO C++, it runs on a 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, we will start our tutorial code with the following code:

#include <iostream>

#include <elfio/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 section of the tutorial will explain how to work with the reader portion of the ELFIO library. The full text of this tutorial comes together with ELFIO library distribution.

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



The load() method returns ‘true’ if corresponding file was found and processed successfully.

ELF file header properties are accessible now. So, we may require ELF 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 ) 1.png

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

else

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

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

if ( reader.get\_encoding() == ELFDATA2LSB ) 2.png

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

else

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

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

2.png - 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 values for 32/64 bit architectures. Constants ELFDATA2LSB and ELFDATA2MSB define values for little- and big-endian encoding.

ELF binary files may consist of several sections. Each section has its own responsibility: some contains executable code; other describe program dependencies; other symbol tables and so on. See ELF binary format documentation for purpose and content 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(); 1.png

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

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

const section\* psec = reader.sections[i]; 2.png

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

}

1.png - Retrieve number of sections

2.png - Use operator[] to access a section by its number or symbolic name

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

‘sections’ data 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(); 1.png

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

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

const segment\* pseg = reader.segments[i]; 2.png

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

}

1.png - Retrieve the number of segments

2.png - 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’ data member of the ‘reader’ class.

## ELF Section Data Accessors

To simplify creation and interpretation of the ELF sections' data, the ELFIO library provides accessor classes. To the moment of writing this document, the following classes are available:

* String section accessor
* Symbol section accessor
* Relocation section accessor
* Note section accessor
* Dynamic section accessor

Definitely, it is possible to extend the library by implementing additional accessors for less generic and customized purposes. More accessors may be implemented in future versions of the library.

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 ) { 1.png

const symbol\_section\_accessor symbols( reader, psec ); 2.png

for ( unsigned int j = 0; j < symbols.get\_symbols\_num(); ++j ) { 3.png

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 ); 4.png

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

}

}

 - Check section’s type

- Build symbol section accessor



 - Get the number of symbols by using the symbol section accessor

 - Get particular symbol properties – its name, value, etc.

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

## ELFDump Utility

The source code for the ELF Dump Utility can be found in the "examples" directory. This utility is heavily relies on dump facilities provided by auxiliary header file <elfio\_dump.hpp>. The header file demonstrates more accessor’s usage examples.

## ELF File Writer

Let’s see how easy to create a new executable ELF file now.

In this chapter will create simple “Hello World” executable file without involving of compiler and/or assembler. The executable file will be created and run on i386 Linux OS platform. It supposed to run well on both 32 and 64-bit Linux platforms.

Before we start, let’s mention one important topic. ELF standard does not require that executable file will contain ELF sections – only presence of ELF segments is required. elfio library designed that way that all data belongs to a section. It means that to make a segment’s data, sections should be created first. Those sections are associated with segment by invocation of segment’s member function add\_section\_index().

Yet another worth mentioning thing is that elfio library creates required string table section automatically, by itself.

Our usage of the library API will consist of several steps:

* Creation of empty elfio object
* Setting-up ELF file properties
* Creation code section and data content for it
* Creation data section and its content
* Addition of both sections to corresponding ELF file segments
* Setting-up program entry point
* Serialization of elfio object to executable ELF file

#include <elfio/elfio.hpp>

using namespace ELFIO;

int main( void )

{

elfio writer;

writer.create( ELFCLASS32, ELFDATA2LSB );

writer.set\_os\_abi( ELFOSABI\_LINUX );

writer.set\_type( ET\_EXEC );

writer.set\_machine( EM\_386 );

section\* text\_sec = writer.sections.add( ".text" );

text\_sec->set\_type( SHT\_PROGBITS );

text\_sec->set\_flags( SHF\_ALLOC | SHF\_EXECINSTR );

text\_sec->set\_addr\_align( 0x10 );

char text[] = { '\xB8', '\x04', '\x00', '\x00', '\x00', // mov eax, 4

'\xBB', '\x01', '\x00', '\x00', '\x00', // mov ebx, 1

'\xB9', '\x20', '\x80', '\x04', '\x08', // mov ecx, msg

'\xBA', '\x0E', '\x00', '\x00', '\x00', // mov edx, 14

'\xCD', '\x80', // int 0x80

'\xB8', '\x01', '\x00', '\x00', '\x00', // mov eax, 1

'\xCD', '\x80' }; // int 0x80

text\_sec->set\_data( text, sizeof( text ) );

segment\* text\_seg = writer.segments.add();

text\_seg->set\_type( PT\_LOAD );

text\_seg->set\_virtual\_address( 0x08048000 );

text\_seg->set\_physical\_address( 0x08048000 );

text\_seg->set\_flags( PF\_X | PF\_R );

text\_seg->set\_align( 0x1000 );

text\_seg->add\_section\_index( text\_sec->get\_index(),

text\_sec->get\_addr\_align() );

section\* data\_sec = writer.sections.add( ".data" );

data\_sec->set\_type( SHT\_PROGBITS );

data\_sec->set\_flags( SHF\_ALLOC | SHF\_WRITE );

data\_sec->set\_addr\_align( 0x4 );

char data[] = { '\x48', '\x65', '\x6C', '\x6C', '\x6F', // “Hello, World!\n”

'\x2C', '\x20', '\x57', '\x6F', '\x72',

'\x6C', '\x64', '\x21', '\x0A' };

data\_sec->set\_data( data, sizeof( data ) );

segment\* data\_seg = writer.segments.add();

data\_seg->set\_type( PT\_LOAD );

data\_seg->set\_virtual\_address( 0x08048020 );

data\_seg->set\_physical\_address( 0x08048020 );

data\_seg->set\_flags( PF\_W | PF\_R );

data\_seg->set\_align( 0x10 );

data\_seg->add\_section\_index( data\_sec->get\_index(),

data\_sec->get\_addr\_align() );

writer.set\_entry( 0x08048000 );

writer.save( "hello\_i386\_32" );

return 0;

}

# ELFIO Library Classes

## elfio

### Data members

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

|  |  |
| --- | --- |
| **Data member** | **Description** |
| Sections | The container stores ELFIO library section instances. Implements operator[] and size(). operator[] permits access to individual ELF file section according to its index. |
| Segments | The container stores ELFIO library segment instances. Implements operator[] and size(). operator[] permits access to individual ELF file segment according to its index. |

### Member functions

Here is the list of elfio public member functions. The functions permit to retrieve or set ELF file properties.

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **elfio**() | The constructor. |
| **~elfio**() | The destructor. |
| void  **create**(  unsigned char file\_class,  unsigned char encoding ) | Cleans and initializes elfio object. *file\_class* is either ELFCLASS32 or ELFCLASS64. *file\_class* is either ELFDATA2LSB or ELFDATA2MSB. |
| bool  **load**(  const std::string& file\_name ) | Initializes elfio object by loading data from ELF binary file. File name provided in *file\_name*.  Returns true if the file was processed successfully. |
| bool  **save**(  const std::string& file\_name ) | Creates a file in ELF binary format. File name provided in *file\_name*. Returns true if the file was created successfully. |
| unsigned char  **get\_class**() | Returns ELF file class. Possible values are ELFCLASS32 or ELFCLASS64. |
| unsigned char  **get\_elf\_version**() | Returns ELF file format version. |
| unsigned char  **get\_encoding**() | Returns ELF file format encoding. Possible values are ELFDATA2LSB and ELFDATA2MSB. |
| Elf\_Word  **get\_version**() | Identifies the object file version. |
| Elf\_Half  **get\_header\_size**() | Returns the ELF header's size in bytes. |
| Elf\_Half  **get\_section\_entry\_size**() | Returns a section's entry size in ELF file header section table. |
| Elf\_Half  **get\_segment\_entry\_size**() | Returns a segment's entry size in ELF file header program table. |
| unsigned char  **get\_os\_abi**() | Returns operating system ABI identification. |
| void  **set\_os\_abi**(  unsigned char *value* ) | Sets operating system ABI identification. |
| unsigned char  **get\_abi\_version**(); | Returns ABI version. |
| void  **set\_abi\_version**(  unsigned char *value* ) | Sets ABI version. |
| Elf\_Half  **get\_type**() | Returns the object file type. |
| void  **set\_type**( Elf\_Half *value* ) | Sets the object file type. |
| Elf\_Half  **get\_machine**() | Returns the object file's architecture. |
| void  **set\_machine**( Elf\_Half *value* ) | Sets the object file's architecture. |
| Elf\_Word  **get\_flags** () | Returns processor-specific flags associated with the file. |
| void  **set\_flags**(Elf\_Word *value* ) | Sets processor-specific flags associated with the file. |
| Elf64\_Addr  **get\_entry**() | Returns the virtual address to which the system first transfers control. |
| void  **set\_entry**( Elf64\_Addr *value* ) | Sets the virtual address to which the system first transfers control. |
| Elf64\_Off  **get\_sections\_offset**() | Returns the section header table's file offset in bytes. |
| void  **set\_sections\_offset**(  Elf64\_Off *value* ) | Sets the section header table's file offset. Attention! The value can be overridden by the library, when it creates new ELF file layout. |
| Elf64\_Off  **get\_segments\_offset**() | Returns the program header table's file offset. |
| void  **set\_segments\_offset**(  Elf64\_Off *value* ) | Sets the program header table's file offset. Attention! The value can be overridden by the library, when it creates new ELF file layout. |
| Elf\_Half  **get\_section\_name\_str\_index**() | Returns the section header table index of the entry associated with the section name string table. |
| void  **set\_section\_name\_str\_index**(  Elf\_Half value ) | Sets the section header table index of the entry associated with the section name string table. |
| endianess\_convertor&  **get\_convertor**() | Returns endianess convertor reference for the specific elfio object instance. |
| Elf\_Xword  **get\_default\_entry\_size**(  Elf\_Word *section\_type* ) | Returns default entry size for known section types having different values on 32 and 64 bit architectures. At the moment, only SHT\_RELA,  SHT\_REL, SHT\_SYMTAB and SHT\_DYNAMIC  are 'known' section types. The function returns 0 for other section types. |

## section

Class ‘section’ has no public data members.

### Member functions

section public member functions listed in the table below. These functions permit to retrieve or set ELF file section properties

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **section**() | The default constructor. |
| **~section**() | The destructor. |

## segment

Class ‘segment’ has no public data members.

### Member functions