



ELFIO

Tutorial and User Manual

Abstract

ELFIO is a header-only C++ library intended for reading and generating files in the ELF binary format

Serge Lamikhov-Center
to_serge@users.sourceforge.net

1 TABLE OF CONTENTS

2	INTRODUCTION	2
3	GETTING STARTED WITH ELFIO	2
3.1	ELF FILE READER	2
3.2	ELF SECTION DATA ACCESSORS	5
3.3	ELFDUMP UTILITY	6
3.4	ELF FILE WRITER	6
4	ELFIO LIBRARY CLASSES	10
4.1	ELFIO	10
4.2	SECTION	12
4.3	SEGMENT	14
4.4	STRING_SECTION_ACCESSOR	15
4.5	SYMBOL_SECTION_ACCESSOR	15
4.6	RELOCATION_SECTION_ACCESSOR	17
4.7	DYNAMIC_SECTION_ACCESSOR	18
4.8	NOTE_SECTION_ACCESSOR	19

2 INTRODUCTION

ELFIO is a header-only C++ library intended for reading and generating files in the ELF binary format. It is used as a standalone library - it is not dependant on any other product or project. Adhering to ISO C++, it compiles on a wide variety of architectures and compilers.

While the library is easy to use, some basic knowledge of the ELF binary format is required. Such Information can easily be found on the Web.

The full text of this tutorial comes together with ELFIO library distribution

3 GETTING STARTED WITH ELFIO

3.1 ELF FILE READER

The ELFIO library is just normal C++ header files. In order to use all its classes and types, simply include the main header file "elfio.hpp". All ELFIO library declarations reside in a namespace called "ELFIO". This can be seen in the following example:

```
#include <iostream>
#include <elfio/elfio.hpp>           ❶

using namespace ELFIO             ❷

int main( int argc, char** argv )
{
    if ( argc != 2 ) {
        std::cout << "Usage: tutorial <elf_file>" << std::endl;
        return 1;
    }
}
```

❶ - Include `elfio.hpp` header file

❷ - The ELFIO namespace usage

This section of the tutorial will explain how to work with the reader portion of the ELFIO library.

The first step would be creating an `elfio` class instance. The `elfio` constructor has no parameters. The creation is normally followed by invoking the 'load' member method, passing it an 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;
}

```

❶ - Create `elfio` class instance

❷ - 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 the corresponding file was found and processed successfully.

All the ELF file header properties such as encoding, machine type and entry point are accessible now. 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;

```

❶ - Member function `get_class()` returns ELF file class. Possible return values are: `ELFCLASS32` or `ELFCLASS64`

❷ - 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 consist of sections and segments. Each section has its own responsibility: some contains executable code, others –program's data, some are symbol tables and so on. See ELF binary format documentation for purpose and content description of sections and segments.

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();           ❶
std::cout << "Number of sections: " << sec_num << std::endl;
for ( int i = 0; i < sec_num; ++i ) {
    const section* psec = reader.sections[i];       ❷
    std::cout << " [" << i << "] "
                << psec->get_name()                 ❸
                << "\t"
                << psec->get_size()                 ❸
                << std::endl;
    // Access section's data
    const char* p = reader.sections[i]->get_data();  ❸
}
```

- ❶ - Retrieve the number of sections
- ❷ - Use operator[] to access a section by its number or symbolic name
- ❸ - get_name(), get_size() and get_data() are member functions of 'section' class

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

Similarly, for executables, the segments of the ELF file can be processed:

```
// Print ELF file segments info
Elf_Half seg_num = reader.segments.size();          ❶
std::cout << "Number of segments: " << seg_num << std::endl;
for ( int i = 0; i < seg_num; ++i ) {
    const segment* pseg = reader.segments[i];       ❷
    std::cout << " [" << i << "] 0x" << std::hex
                << pseg->get_flags()                 ❸
                << "\t0x"
                << pseg->get_virtual_address()       ❸
                << "\t0x"
                << pseg->get_file_size()             ❸
                << "\t0x"
                << pseg->get_memory_size()           ❸
                << std::endl;
    // Access segments's data
    const char* p = reader.segments[i]->get_data();  ❸
}
```

- ❶ - Retrieve the number of segments
- ❷ - Use operator[] to access a segment by its number
- ❸ - get_flags(), get_virtual_address(), get_file_size(), get_memory_size() and get_data() are member methods of 'segment' class

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

3.2 ELF SECTION DATA ACCESSORS

To simplify creation and interpretation of specific ELF sections, the ELFIO library provides accessor classes. Currently, the following classes 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.

Let's see how the accessors can be used with the previous ELF file reader example. The following example prints out all symbols in a section that turns out to be a symbol section:

```
if ( psec->get_type() == SHT_SYMTAB ) { ❶  
    const symbol_section_accessor symbols( reader, psec ); ❷  
    for ( unsigned int j = 0; j < symbols.get_symbols_num(); ++j ) { ❸  
        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 ); ❹  
        std::cout << j << " " << name << std::endl;  
    }  
}
```

- ❶ - Check section's type
- ❷ - Build symbol section accessor
- ❸ - Get the number of symbols by using the symbol section accessor
- ❹ - Get particular symbol properties – its name, value, etc.

First we create a 'symbol_section_accessor' class instance. Usually, accessors's constructors receive references to both the `elfio` and a 'section' objects as parameters. The `get_symbol()` method is used for retrieving particular entries in the symbol table.

3.3 ELFDUMP UTILITY

The source code for the ELF Dump Utility can be found in the "examples" directory. It heavily relies on dump facilities provided by the auxiliary header file `<elfio_dump.hpp>`. This header file demonstrates more accessor's usage examples.

3.4 ELF FILE WRITER

In this chapter we will create a simple ELF executable file that prints out the classical "Hello, World!" message. The executable will be created and run on i386 Linux OS platform. It is supposed to run well on both 32 and 64-bit Linux platforms. The file will be created without invoking the compiler or assembler tools in the usual way (i.e. translating high level source code that makes use of the standard library functions). Instead, using the ELFIO writer, all the necessary sections and segments of the file will be created and filled explicitly, each, with its appropriate data. The physical file would then be created by the ELFIO library.

Before starting, two implementation choices of `elfio` that users should be aware of are:

1. The ELF standard does not require that executables will contain any ELF sections – only presence of ELF segments is mandatory. The `elfio` library, however, requires that all data will belong to sections. It means that in order to put data in a segment, a section should be created first. Sections are associated with segments by invoking the segment's member function `add_section_index()`.
2. The `elfio` writer class, while constructing, creates a string table section automatically.

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

- Creating an empty `elfio` object
- Setting-up ELF file properties
- Creating code section and data content for it
- Creating data section and its content
- Addition of both sections to corresponding ELF file segments
- Setting-up the program's entry point
- Dumping the `elfio` object to an 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;
}

```


-
- ❶ - Initialize empty 'elfio' object. This should be done as the first step when creating a new 'elfio' object as other API is relying on parameters provided – ELF file 32-bits/64-bits and little/big endianness
 - ❷ - Other attributes of the file. Linux OS loader does not require full set of the attributes, but they are provided when a regular linker used for creation of ELF files
 - ❸ - Create a new section, set section's attributes. Section type, flags and alignment have a big significance and controls how this section is treated by a linker or OS loader
 - ❹ - Add section's data
 - ❺ - Create new segment
 - ❻ - Set attributes and properties for the segment
 - ❼ - Associate a section with segment containing it
 - ❽ - Setup entry point for your program
 - ❾ - Create ELF binary file on disk

Let's compile the example above (put into a source file named 'writer.cpp') into an executable file (named 'writer'). Invoking 'writer' will create the executable file "hello_i386_32" that prints the "Hello, World!" message. We'll change the permission attributes of this file, and finally, run it:

```
> ls
writer.cpp
> g++ writer.cpp -o writer
> ls
writer  writer.cpp
> ./writer
> ls
hello_i386_32  writer  writer.cpp
> chmod +x ./hello_i386_32
> ./hello_i386_32
Hello, World!
```

In case you already compiled the 'elfdump' utility, you can inspect the properties of the produced executable file (the '.note' section was not discussed in this tutorial, but it is produced by the sample file writer.cpp located in the 'examples' folder of the library distribution):

```
./elfdump hello_i386_32
```

ELF Header

```
Class:      ELF32
Encoding:   Little endian
ELFVersion: Current
Type:       Executable file
Machine:    Intel 80386
Version:    Current
Entry:      0x8048000
Flags:      0x0
```

Section Headers:

[Nr]	Type	Addr	Size	ES	Flg	Lk	Inf	Al	Name
[0]	NULL	00000000	00000000	00		0	0	0	
[1]	STRTAB	00000000	0000001d	00		0	0	0	.shstrtab
[2]	PROGBITS	08048000	0000001d	00	AX	0	0	16	.text
[3]	PROGBITS	08048020	0000000e	00	WA	0	0	4	.data
[4]	NOTE	00000000	00000044	00		0	0	1	.note

Key to Flags: W (write), A (alloc), X (execute)

Segment headers:

[Nr]	Type	VirtAddr	PhysAddr	FileSize	Mem.Size	Flags	Align
[0]	LOAD	08048000	08048000	0000001d	0000001d	RX	00001000
[1]	LOAD	08048020	08048020	0000000e	0000000e	RW	00000010

Note section (.note)

No	Type	Name
[0]	00000001	Created by ELFIO
[1]	00000001	Never easier!

Note:

The `elfio` library takes care of the resulting binary file layout calculation. It does this on base of the provided memory image addresses and sizes. It is the user's responsibility to provide correct values for these parameters. Please refer to your OS (other execution environment or loader) manual for specific requirements related to executable ELF file attributes and/or mapping.

Similarly to the 'reader' example, you may use provided accessor classes to interpret and modify content of section's data.

4 ELFIO LIBRARY CLASSES

This section contains detailed description of classes provided by `elfio` library

4.1 ELFIO

4.1.1 Data members

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

Data member	Description
<code>sections</code>	The container stores ELFIO library section instances. Implements <code>operator[]</code> , <code>add()</code> and <code>size()</code> . <code>operator[]</code> permits access to individual ELF file section according to its index or section name.
<code>segments</code>	The container stores ELFIO library segment instances. Implements <code>operator[]</code> , <code>add()</code> and <code>size()</code> . <code>operator[]</code> permits access to individual ELF file segment according to its index.

4.1.2 Member functions

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

Member Function	Description
<code>elfio()</code>	The constructor.
<code>~elfio()</code>	The destructor.
<code>void create(unsigned char file_class, unsigned char encoding)</code>	Cleans and/or initializes <code>elfio</code> object. <i>file_class</i> is either <code>ELFCLASS32</code> or <code>ELFCLASS64</code> . <i>file_class</i> is either <code>ELFDATA2LSB</code> or <code>ELFDATA2MSB</code> .
<code>bool load(const std::string& file_name)</code>	Initializes <code>elfio</code> object by loading data from ELF binary file. File name provided in <i>file_name</i> . Returns true if the file was processed successfully.
<code>bool save(const std::string& file_name)</code>	Creates a file in ELF binary format. File name provided in <i>file_name</i> . Returns true if the file was created successfully.
<code>unsigned char get_class()</code>	Returns ELF file class. Possible values are <code>ELFCLASS32</code> or <code>ELFCLASS64</code> .

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 <i>value</i>)	Sets operating system ABI identification.
unsigned char get_abi_version();	Returns ABI version.
void set_abi_version (unsigned char <i>value</i>)	Sets ABI version.
Elf_Half get_type()	Returns the object file type.
void set_type (Elf_Half <i>value</i>)	Sets the object file type.
Elf_Half get_machine()	Returns the object file's architecture.
void set_machine (Elf_Half <i>value</i>)	Sets the object file's architecture.
Elf_Word get_flags ()	Returns processor-specific flags associated with the file.
void set_flags (Elf_Word <i>value</i>)	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 <i>value</i>)	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 <i>value</i>)	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 <i>value</i>)	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 <i>value</i>)	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 <code>elfio</code> object instance.
Elf_Xword get_default_entry_size (Elf_Word <i>section_type</i>)	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.

4.2 SECTION

Class 'section' has no public data members.

4.2.1 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
-----------------	-------------

<code>section()</code>	The default constructor. No section class instances are created manually. Usually, 'add' method is used for 'sections' data member of 'elfio' object
<code>~section()</code>	The destructor.
Elf_Half <code>get_index()</code>	Returns section index. Sometimes, this index is passed to another section for inter-referencing between the sections. Section's index is also passed to 'segment' for segment/section association
Set functions: void <code>set_name</code> (std::string) void <code>set_type</code> (Elf_Word) void <code>set_flags</code> (Elf_Xword) void <code>set_info</code> (Elf_Word) void <code>set_link</code> (Elf_Word) void <code>set_addr_align</code> (Elf_Xword) void <code>set_entry_size</code> (Elf_Xword) void <code>set_address</code> (Elf64_Addr) void <code>set_size</code> (Elf_Xword) void <code>set_name_string_offset</code> (Elf_Word)	Sets attributes for the section
Get functions: std::string <code>get_name</code> () Elf_Word <code>get_type</code> () Elf_Xword <code>get_flags</code> () Elf_Word <code>get_info</code> () Elf_Word <code>get_link</code> () Elf_Xword <code>get_addr_align</code> () Elf_Xword <code>get_entry_size</code> () Elf64_Addr <code>get_address</code> () Elf_Xword <code>get_size</code> () Elf_Word <code>get_name_string_offset</code> ()	Returns section attributes
Data manipulation functions: const char* <code>get_data</code> () void <code>set_data</code> (const char* pData, Elf_Word size) void <code>set_data</code> (const std::string& data)	Manages section data

<pre>void append_data (const char* pData, Elf_Word size) void append_data (const std::string& data)</pre>	
--	--

4.3 SEGMENT

Class 'segment' has no public data members.

4.3.1 Member functions

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

Member Function	Description
segment ()	The default constructor. No segment class instances are created manually. Usually, 'add' method is used for 'segments' data member of 'elfio' object
~segment ()	The destructor.
Elf_Half get_index ()	Returns segment's index
Set functions: void set_type (Elf_Word) void set_flags (Elf_Word) void set_align (Elf_Xword) void set_virtual_address (Elf64_Addr) void set_physical_address (Elf64_Addr) void set_file_size (Elf_Xword) void set_memory_size (Elf_Xword)	Sets attributes for the segment
Get functions: Elf_Word get_type () Elf_Word get_flags () Elf_Xword get_align () Elf64_Addr get_virtual_address () Elf64_Addr get_physical_address () Elf_Xword get_file_size () Elf_Xword get_memory_size ()	Returns segment attributes

<pre>Elf_Half add_section_index(Elf_Half index, Elf_Xword addr_align) Elf_Half get_sections_num() Elf_Half get_section_index_at(Elf_Half num)</pre>	Manages segment-section association
--	-------------------------------------

4.4 STRING_SECTION_ACCESSOR

4.4.1 Member functions

Member Function	Description
<pre>string_section_accessor(section* section_)</pre>	The constructor
<pre>const char* get_string(Elf_Word index)</pre>	Retrieves string by its offset (index) in the section
<pre>Elf_Word add_string(const char* str) Elf_Word add_string(const std::string& str)</pre>	Appends section data with new string. Returns position (index) of the new record

4.5 SYMBOL_SECTION_ACCESSOR

4.5.1 Member functions

Member Function	Description
<pre>symbol_section_accessor(const elfio& elf_file, section* symbols_section)</pre>	The constructor
<pre>Elf_Half get_index()</pre>	Returns segment's index

<pre>Elf_Xword get_symbols_num()</pre>	Returns number of symbols in the section
<pre>Get functions: bool get_symbol(Elf_Xword index, std::string& name, Elf64_Addr& value, Elf_Xword& size, unsigned char& bind, unsigned char& type, Elf_Half& section_index, unsigned char& other) bool get_symbol(const std::string& name, Elf64_Addr& value, Elf_Xword& size, unsigned char& bind, unsigned char& type, Elf_Half& section_index, unsigned char& other)</pre>	Retrieves symbol properties by symbol index or name
<pre>Elf_Word add_symbol(Elf_Word name, Elf64_Addr value, Elf_Xword size, unsigned char info, unsigned char other, Elf_Half shndx) Elf_Word add_symbol(Elf_Word name, Elf64_Addr value, Elf_Xword size, unsigned char bind, unsigned char type, unsigned char other, Elf_Half shndx) Elf_Word add_symbol(string_section_accessor& pStrWriter, const char* str, Elf64_Addr value,</pre>	Adds symbol to the symbol table updating corresponding string section if required

<pre> Elf_Xword size, unsigned char info, unsigned char other, Elf_Half shndx) Elf_Word add_symbol(string_section_accessor& pStrWriter, const char* str, Elf64_Addr value, Elf_Xword size, unsigned char bind, unsigned char type, unsigned char other, Elf_Half shndx) </pre>	
--	--

4.6 RELOCATION_SECTION_ACCESSOR

4.6.1 Member functions

Member Function	Description
<pre> relocation_section_accessor(const elfio& elf_file_, section* section_) </pre>	The constructor
<pre> Elf_Xword get_entries_num() </pre>	Retrieves number of relocation entries in the section
<pre> bool get_entry(Elf_Xword index, Elf64_Addr& offset, Elf_Word& symbol, Elf_Word& type, Elf_Sxword& addend) bool get_entry(Elf_Xword index, Elf64_Addr& offset, Elf64_Addr& symbolValue, std::string& symbolName, Elf_Word& type, Elf_Sxword& addend, Elf_Sxword& calcValue) </pre>	Retrieves properties for relocation entry by its index. Calculated value in the second flavor of this function may not work for all architectures

<pre> void add_entry(Elf64_Addr offset, Elf_Xword info) void add_entry(Elf64_Addr offset, Elf_Word symbol, unsigned char type) void add_entry(Elf64_Addr offset, Elf_Xword info, Elf_Sxword addend) void add_entry(Elf64_Addr offset, Elf_Word symbol, unsigned char type, Elf_Sxword addend) void add_entry(string_section_accessor str_writer, const char* str, symbol_section_accessor sym_writer, Elf64_Addr value, Elf_Word size, unsigned char sym_info, unsigned char other, Elf_Half shndx, Elf64_Addr offset, unsigned char type) </pre>	<p>Adds new relocation entry. The last function in this set is capable to add relocation entry for a symbol, automatically updating symbol and string tables for this symbol</p>
---	--

4.7 DYNAMIC_SECTION_ACCESSOR

4.7.1 Member functions

Member Function	Description
<pre> dynamic_section_accessor (elfio& elf_file_, section* section_) </pre>	<p>The constructor</p>

Elf_Xword get_entries_num()	Retrieves number of dynamic section entries in the section
bool get_entry (Elf_Xword index, Elf_Xword& tag, Elf_Xword& value, std::string& str)	Retrieves properties for dynamic section entry by its index. For most entries only <code>tag</code> and <code>value</code> arguments are relevant. <code>str</code> argument is empty string in this case. If <code>tag</code> equal to <code>DT_NEEDED</code> , <code>DT_SONAME</code> , <code>DT_RPATH</code> or <code>DT_RUNPATH</code> , <code>str</code> argument is filled with value taken from dynamic string table section.
void add_entry (Elf_Xword& tag, Elf_Xword& value) void add_entry (Elf_Xword& tag, std::string& str)	Adds new dynamic section entry. The second variant of the function updates the dynamic string table updating the entry with string table index.

4.8 NOTE_SECTION_ACCESSOR

4.8.1 Member functions

Member Function	Description
note_section_accessor (const elfio& elf_file_, section* section_)	The constructor
Elf_Word get_notes_num()	Retrieves number of note entries in the section
bool get_note (Elf_Word index, Elf_Word& type, std::string& name, void*& desc, Elf_Word& descSize)	Retrieves particular note by its index
void add_note (Elf_Word type, const std::string& name, const void* desc, Elf_Word descSize)	Appends the section with a new note