Serge Lamikhov-Center

to\_serge@users.sourceforge.net

Abstract

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

ELFIO

Tutorial and User Manual

Table of Contents

[2 Introduction 2](#_Toc49068223)

[3 Getting Started With ELFIO 2](#_Toc49068224)

[3.1 ELF File Reader 2](#_Toc49068225)

[3.2 ELF Section Data Accessors 5](#_Toc49068226)

[3.3 elfdump Utility 6](#_Toc49068227)

[3.4 ELF File Writer 6](#_Toc49068228)

[4 ELFIO Library Classes 10](#_Toc49068229)

[4.1 elfio 10](#_Toc49068230)

[4.2 section 13](#_Toc49068231)

[4.3 segment 14](#_Toc49068232)

[4.4 string\_section\_accessor 15](#_Toc49068233)

[4.5 symbol\_section\_accessor 16](#_Toc49068234)

[4.6 relocation\_section\_accessor 18](#_Toc49068235)

[4.7 dynamic\_section\_accessor 19](#_Toc49068236)

[4.8 note\_section\_accessor 20](#_Toc49068237)

[4.9 modinfo\_section\_accessor 21](#_Toc49068238)

# 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

# Getting Started With ELFIO

## 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> 

using namespace ELFIO 

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

{

if ( argc != 2 ) {

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

return 1;

}

1.png - Include elfio.hpp header file

2.png - 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;

}

 - 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 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 ) 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 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(); 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 the 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

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(); 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, the segments' attributes and data are obtained by using the ‘segments’ data member of ELFIO's ‘reader’ class.

## 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 ) { 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.

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.

## 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.

## 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.

**Note:**

The example below demonstrates creation of 32-bit architecture executable file. The source file in the ‘examples’ directory was modified to generate 64-bit Linux executable file.

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 ); 1.png

writer.set\_os\_abi( ELFOSABI\_LINUX ); 2.png

writer.set\_type( ET\_EXEC );

writer.set\_machine( EM\_386 );

section\* text\_sec = writer.sections.add( ".text" ); 3.png

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 ) ); 4.png

segment\* text\_seg = writer.segments.add(); 5.png

text\_seg->set\_type( PT\_LOAD ); 6.png

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(), 7.png

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

section\* data\_sec = writer.sections.add( ".data" ); 3.png

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 ) ); 4.png

segment\* data\_seg = writer.segments.add(); 5.png

data\_seg->set\_type( PT\_LOAD ); 6.png

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(), 7.png

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

writer.set\_entry( 0x08048000 ); 8.png

writer.save( "hello\_i386\_32" ); 9.png

return 0;

}

1.png - 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

2.png - 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

- 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



- Add section’s data



5.png - Create new segment

6.png - Set attributes and properties for the segment

7.png - Associate a section with segment containing it

8.png - Setup entry point for your program

9.png - 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.

# ELFIO Library Classes

This section contains detailed description of classes provided by elfio library

## 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[], add() and size(). operator[] permits access to individual ELF file section according to its index or section name.  operator[] is capable to provide section pointer according to section index or section name. begin() and end() iterators are available too. |
| segments | The container stores ELFIO library segment instances. Implements operator[], add() and size(). operator[] permits access to individual ELF file segment according to its index.  operator[] is capable to provide section pointer according to segment index. begin() and end() iterators are available too. |

### 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/or initializes elfio object. *file\_class* is either ELFCLASS32 or ELFCLASS64. *file\_class* is either ELFDATA2LSB or ELFDATA2MSB. |
| bool  **load**(  const std::string& file\_name )  bool  **load**( std::istream &stream ) | Initializes elfio object by loading data from ELF binary file. File name is provided as a std::string in *file\_name* or as an opened std::istream in *stream*.  Returns true if the file was processed successfully. |
| bool  **save**(  const std::string& file\_name )  bool  **save**( std::ostream &stream ) | Creates a file in ELF binary format. File name is provided as a std::string in *file\_name* or as an opened std::ostream in *stream*.  Returns true if the file has been 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/executable file version. |
| void  **set\_version**( Elf\_Word value ) | Sets the object/executable 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. No section class instances are created manually. Usually, ‘add’ method is used for ‘sections’ data member of ‘elfio’ object |
| **~section**() | The destructor. |
| Elf\_Half  **get\_index()** | 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 **set\_name**( std::string )  void **set\_type**( Elf\_Word )  void **set\_flags**( Elf\_Xword )  void **set\_info**( Elf\_Word )  void **set\_link**( Elf\_Word )  void **set\_addr\_align**( Elf\_Xword )  void **set\_entry\_size**( Elf\_Xword )  void **set\_address**( Elf64\_Addr )  void **set\_size**( Elf\_Xword )  void **set\_name\_string\_offset**( Elf\_Word ) | Sets attributes for the section |
| Get functions:  std::string **get\_name**()  Elf\_Word **get\_type**()  Elf\_Xword **get\_flags**()  Elf\_Word **get\_info**()  Elf\_Word **get\_link**()  Elf\_Xword **get\_addr\_align**()  Elf\_Xword **get\_entry\_size**()  Elf64\_Addr **get\_address**()  Elf\_Xword **get\_size**()  Elf\_Word **get\_name\_string\_offset**() | Returns section attributes |
| Data manipulation functions:  const char\* **get\_data**()  void **set\_data**(  const char\* pData,  Elf\_Word size )  void **set\_data**(  const std::string& data )  void **append\_data**(  const char\* pData,  Elf\_Word size )  void **append\_data**(  const std::string& data ) | Manages section data |

## segment

Class ‘segment’ has no public data members.

### 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 |
| 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 ) | Manages segment-section association |
| const char\* **get\_data**() | Provides content of segment’s data |

## string\_section\_accessor

### Member functions

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **string\_section\_accessor**(  section\* section\_ ) | The constructor |
| const char\*  **get\_string**(  Elf\_Word index ) | Retrieves string by its offset (index) in the section |
| Elf\_Word  **add\_string**(  const char\* str )  Elf\_Word  **add\_string**(  const std::string& str ) | Appends section data with new string. Returns position (index) of the new record |

## symbol\_section\_accessor

### Member functions

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **symbol\_section\_accessor**(  const elfio& elf\_file,  section\* symbols\_section ) | The constructor |
| Elf\_Half  **get\_index()** | Returns segment’s index |
| Elf\_Xword  **get\_symbols\_num**() | Returns number of symbols in the section |
| 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 )  bool  **get\_symbol**(  const Elf64\_Addr& value,  std::string& name,  Elf\_Xword& size,  unsigned char& bind,  unsigned char& type,  Elf\_Half& section\_index,  unsigned char& other ) | Retrieves symbol properties by symbol index, name or address |
| 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,  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 ) | Adds symbol to the symbol table updating corresponding string section if required |
| Elf\_Xword  **arrange\_local\_symbols**(    std::function<void(  Elf\_Xword first,  Elf\_Xword second )> func = nullptr) | ELF standard requires that symbols with STB\_LOCAL binding will be ordered prior any other entries in the symbol table.  The function rearranges the symbols and invokes a callback for each swap between symbols.  Also see **swap\_symbols**() function of relocation section accessor |

## relocation\_section\_accessor

### Member functions

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **relocation\_section\_accessor**(  const elfio& elf\_file\_,  section\* section\_ ) | The constructor |
| Elf\_Xword  **get\_entries\_num**() | Retrieves number of relocation entries in the section |
| 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 ) | Retrieves properties for relocation entry by its index. Calculated value in the second flavor of this function may not work for all architectures |
| 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 ) | 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 |
| Void  **swap\_symbols**(  Elf\_Xword first,  Elf\_Xword second ) | A helper function that changes (swaps) symbol numbers in relocation entries.  The function can be used as a callback for **arrange\_local\_symbols().** |

## dynamic\_section\_accessor

### Member functions

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **dynamic\_section\_accessor** (  elfio& elf\_file\_,  section\* section\_ ) | The constructor |
| 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 tag and value arguments are relevant. str argument is empty string in this case. If tag equal to DT\_NEEDED, DT\_SONAME, DT\_RPATH or DT\_RUNPATH, str 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. |

## note\_section\_accessor

### 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 |

## modinfo\_section\_accessor

### Member functions

|  |  |
| --- | --- |
| **Member Function** | **Description** |
| **modinfo\_section\_accessor**(  section\* section\_ ) | The constructor |
| Elf\_Word  **get\_attribute\_num**() | Retrieves number of attributes in the section |
| bool  **get\_attribute**(  Elf\_Word no,  std::string& field,  std::string& value )  bool  **get\_attribute**(  std::string field,  std::string& value ) | Retrieves attribute by its index or field name |
| Elf\_Word  **add\_attribute**(  std::string field,  std::string value ) | Appends the section with a new attribute |