ELFIO

User's Guide

Contents

List of Tables

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.

Chapter 1

Getting Started With ELFIO

1.1 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 from the following code:

```
#include <iostream>
#include <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 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 that, we initialize the instance by loading an ELF file with name passed as a parameter.

```
    // Create an elfio reader
    elfio reader;                                       ❶

    // Load ELF data
    if ( !reader.load( argv[1] ) ) {                    ❷
        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.

From here, ELF header properties are accessible. 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 )                    ❶
    std::cout << "ELF32" << std::endl;
else
    std::cout << "ELF64" << std::endl;

std::cout << "ELF file encoding : ";
if ( reader.get_encoding() == ELFDATA2LSB )                ❷
    std::cout << "Little endian" << std::endl;
else
    std::cout << "Big endian" << std::endl;
```

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

❷ Member function `get_encoding()` returns ELF file format encoding. Possible values are `ELFDATA2LSB` and `ELFDATA2MSB`.

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. `ELFDATA2LSB` and `ELFDATA2MSB` constants 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()
              << "\t"
              << psec->get_size()
              << std::endl;
    // Access to section's data
    // const char* p = reader.sections[i]->get_data()
}
```

`sections` member of `reader` object permits to obtain number of sections the ELF file contains. 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()
                << "\t0x"
                << pseg->get_virtual_address()
                << "\t0x"
                << pseg->get_file_size()
                << "\t0x"
                << pseg->get_memory_size()
                << std::endl;
        // Access to segments's data
        // const char* p = reader.segments[i]->get_data()
    }
```

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

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

1.2 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`

Definitely, it is possible to extend the library by implementing additional accessors serving particular purposes.

Let's see how the accessors can be used with the previous ELF file reader example. For this example purposes, we will print out all symbols in a 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 create `symbol_section_accessor` instance first. Usually, accessors receive the `elfio` and `section*` parameters for their constructors. `get_symbol` is used to retrieve a particular entry in the symbol table.

1.3 ELFDump Utility

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

1.4 ELF File Writer

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, our tutorial code starts from this:

Chapter 2

ELFIO Library Classes

2.1 Class `elfio`

2.1.1 Data members

The ELFIO library consists of two independent parts: ELF File Reader

Function	Description
sections	A container that stores pointers to ELF file section instances. implementing operator[] and size() f
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.

Table 2.1: Class `elfio` member functions

2.1.2 Member functions

The ELFIO library consists of two independent parts: ELF File Reader (`IELFI`) and ELF Producer (`IELFO`). Each is represented by its own set of interfaces. The library does not contain any classes that need to be explicitly instantiated. ELFIO itself provides the interfaces that are used to access the library's functionality.

Function	Description
elfio (void);	The constructor.
~elfio (void);	The destructor.
void **create** (unsigned char file_class , unsigned char encoding);	Cleans and initializes empty `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** (void);	Returns ELF file class. Possible values are ELFCLASS32 or ELFCLASS64.
unsigned char **get_elf_version** (void);	Returns ELF file format version.
unsigned char **get_encoding** (void);	Returns ELF file format encoding. Possible values are ELFDATA2LSB and ELFDATA2MSB.
Elf_Word **get_version** (void);	Identifies the object file version.
Elf_Half **get_header_size** (void);	Returns the ELF header's size in bytes.
Elf_Half **get_section_entry_size** (void);	Returns a section's entry size in ELF file header section table.
Elf_Half **get_segment_entry_size** (void);	Returns a segment's entry size in ELF file header program table.
unsigned char **get_os_abi** (void);	Returns operating system ABI identification.
void **set_os_abi** (unsigned char value);	Sets operating system ABI identification.
unsigned char **get_abi_version** (void);	Returns ABI version.
void **set_abi_version** (unsigned char value);	Sets ABI version.
Elf_Half **get_type** (void);	Returns the object file type.
void **set_type** (Elf_Half value);	Sets the object file type.
Elf_Half **get_machine** (void);	Returns the object file's architecture.
void **set_machine** (Elf_Half value);	Sets the object file's architecture.
Elf_Word **get_flags** (void);	Returns processor-specific flags associated with the file.
void **set_flags** (Elf_Word value);	Sets processor-specific flags associated with the file.

Table 2.2: Class `elfio` member functions

Function	Description
Elf64_Addr **get_entry** (void);	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** (void);	Returns the section header table's file offset in bytes.
void **set_sections_offset** (Elf64_Off value);	Returns default entry size for .
Elf64_Off **get_segments_offset** (void);	Returns the program header table's file offset in bytes.
void **set_segments_offset** (Elf64_Off value);	Sets the program header table's file offset in bytes.
Elf_Half **get_section_name_str_index** (void);	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** (void);	Returns endianess convertor reference for the specific `elfio` object instance.

Table 2.3: Class `elfio` member functions (continue)

Chapter 3

`IELFO` - ELF File Producer Interface

The ELFIO library can help you build a very short ELF executable file. This chapter shows how to build an executable file that will run on x86 Linux machines and print "Hello World!" on your console.

Just as with the reader, the first step is to get a pointer onto the ELF File Writer (Producer):

```
IELFO* pELFO;
ELFIO::GetInstance()->CreateELFO( &pELFO );
```

Before continuing, the library must be informed about the main attributes of the executable file to be built. To do this, declare that the executable ELF file will run on a 32 bit x86 machine; has little endian encoding and uses the current version of the ELF file format:

```
// You can't proceed without this function call!
pELFO->SetAttr( ELFCLASS32, ELFDATA2LSB, EV_CURRENT,
                ET_EXEC, EM_386, EV_CURRENT, 0 );
```

Some sections of an ELF executable file should reside in the program segments. To create this loadable segment call the `AddSegment()` function.

```
// Create a loadable segment
IELFOSegment* pSegment = pELFO->AddSegment( PT_LOAD,
                                            0x08040000,
                                            0x08040000,
                                            PF_X | PF_R,
                                            0x1000 );
```

The following segment serves as a placeholder for our code section. To create this code section call the AddSection() function:

```
// Create code section
IELFOSection* pTextSec = pELFO->AddSection( ".text",
                                SHT_PROGBITS,
                                SHF_ALLOC | SHF_EXECINSTR,
                                0,
                                0x10,
                                0 );
```

Then, add the executable code for the section:

```
char text[] =
```

```
    { '\xB8', '\x04', '\x00', '\x00', '\x00',   // mov eax, 4
      '\xBB', '\x01', '\x00', '\x00', '\x00',   // mov ebx, 1
      '\xB9', '\xFD', '\x00', '\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
      '\x48', '\x65', '\x6C', '\x6C', '\x6F',   // db 'Hello'
      '\x2C', '\x20', '\x57', '\x6F', '\x72',   // db ', Wor'
      '\x6C', '\x64', '\x21', '\x0A'            // db 'ld!', 10
    };
    pTextSec->SetData( text, sizeof( text ) );
```

Next, this code section is put into the loadable segment:

```
    // Add code section into program segment
    pSegment->AddSection( pTextSec );
    pTextSec->Release();
    pSegment->Release();
```

Finally, define the start address of the program and create the result file:

```
    // Set program entry point
    pELFO->SetEntry( 0x08040000 );
    // Create ELF file
    pELFO->Save( "test.elf" );
    pELFO->Release();
```

Please note: Call the `Release()` functions for each interface you have used. This will free all resources the ELFIO library has created.

Now compile the program and run it. The result is a new ELF file called "test.elf". The size of this working executable file is only 267 bytes! Run it on your Linux machine with the following commands:

```
    [Writer]$ ./Writer
    [Writer]$ chmod +x test.elf
    [Writer]$ ./test.elf
    Hello, World!
```

The full text for this program can be found in the "Writer" directory. Also, in the "Examples" directory, two other programs "WriteObj" and "WriteObj2" demonstrate the creation of ELF object files.