Instructions for objconv

A utility for cross-platform development of function libraries, for converting and modifying object files and for dumping and disassembling object and executable files.

By Agner Fog. © 2007. GNU General Public License.
Version 1.00.

Contents

1 Introduction

Objconv is a utility for facilitating cross-platform development of function libraries, for converting and disassembling object files, and for other development purposes. The latest version of objconv is available at www.agner.org/optimize.

Objconv can perform the following tasks:

- Convert object files between different formats used on different x86 and x86-64 platforms.
- Change symbol names in object files.
- Build, manage and convert static link libraries in various formats for different x86 and x86-64 platforms.
- Dump file headers and other contents of object files, static and dynamic library files, and executable files.
- Disassemble object files and executable files and check instruction code syntax.

The following platforms are supported:

- Windows, 32 and 64 bit x86.
- Linux, 32 and 64 bit x86.
- BSD, 32 and 64 bit x86.
- Mac OS X, Intel based, 32 bit

The source code for objconv can be compiled and run under any of these platforms. The program is compatible with standard make utilities.

Note that objconv is intended for programming experts. It is far from fool proof, and you need to have a very good understanding of how compilers and linkers work in order to use this program. Please do not send your programming questions to me - you will not get any answer.

1.1 File types

An executable file is a file containing machine code that can be executed. This can be a program file or a dynamic link library, also called shared object. The name shared object is used only in Unix-like systems, such as Linux, BSD and Mac OS X.

An object file is an intermediate file used in the building of an executable file. It contains part of the code that will make up the final executable file. An object file usually contains cross-references to functions in other object files.

A static link library means a collection of object files. This is called a static linking library file in Windows terminology or an archive in Unix terminology. I prefer to use the name library because an archive can also mean a .zip or .tar file.

Objconv cannot modify or convert executable files, including dynamic link libraries or shared objects, but it can dump or disassemble such files.

The following table summarizes the type of operations that objconv can do on various file types:

File type and format	Word size, bits	Extension	Operating system	Convert from	Convert to	Modify	Dump	Disassemble
Object file COFF/PE	32, 64	.obj	Windows	x	x	x	x	x
Library file COFF/PE	32, 64	.lib	Windows	x	x	x	x	x
DLL, driver COFF/PE	32, 64	.dll, .sys	Windows	-	-	-	x	x
Executable file COFF/PE	32, 64	.exe	Windows	-	-	-	x	x
Object file OMF	16	.obj	DOS, Windows 3.x	-	-	-	x	x
Object file OMF	32	.obj	Windows	x	x	x	x	x
Library file OMF	16	.lib	DOS, Windows 3.x	-	-	x	x	x
Library file OMF	32	.lib	Windows	x	x	x	x	x
Executable file 16 bit	16	.exe	DOS, Windows 3.x	-	-	-	-	-
Object file ELF	32, 64	.o	Linux, BSD	x	x	x	x	x
Library file ELF	32, 64	.a	Linux, BSD	x	x	x	x	x
Shared Object ELF	32, 64	.so	Linux, BSD	-	-	-	x	x
Executable file ELF	32, 64		Linux, BSD	-	-	-	x	x
Object file Mach-O	32	.o	Mac OS X	-	x	-	x	x
Library file Mach-O	32	.a	Mac OS X	-	x	x	x	x
Shared object Mach-O	32	.so	Mac OS X	-	-	-	x	x
Executable file Mach-O	32		Mac OS X	-	-	-	x	x
Universal binary	32, 64		Mac OS X	-	-	-	x	x

2 Command line syntax

If you want to run objconv under Linux, BSD or Mac (Intel based), then you have to first build the executable. Unpack `source.zip` to a temporary directory and run the build script `build.sh`. To run objconv under Windows, you can just use the executable `objconv.exe`.

Objconv is executed from a command line or from a make utility. The syntax is as follows:

```
objconv options inputfile [outputfile]
```

Options start with a dash `-`. A slash `/` is accepted instead of `-` when running under Windows. Options must be separated by spaces. The order of the options is arbitrary, but all

options must come before `inputfile`. The name of the output file must be different from the input file, except when adding object files to a library file. The option letters are case insensitive, file names and symbol names are case sensitive.

The return value from objconv is zero on success, and equal to the highest error number in case of error. This will stop a make utility in case of error messages, but not in case of warning messages.

Summary of options

`-fXXX`	Convert file to format `XXX`. `XXX` = `COFF`, `OMF`, `ELF`, `MAC`, or `ASM`. `PE` is accepted as a synonym for `COFF`. The word size, 32 or 64, may be appended to the name, e.g. `ELF64`. The output format must be specified, except for the dump command `-fd`, which has no output file.
`-fdXXX`	Dump contents of file. `XXX` can be one or more of the following: `f`: file header, `h`: section headers, `s`: symbol table, `r`: relocation table, `n`: string table (all names).
`-ds`	Strip debug information from file. (Default when converting to a different format).
`-dp`	Preserve debug information, even if it is incompatible with the target system.
`-xs`	Strip exception handling information and other incompatible info. (Default when converting to a different format).
`-xp`	Preserve exception handling information and other incompatible info.
`-nu`	Change leading underscores on symbol names to the default for the target system.
`-nu-`	Remove leading underscores from symbol names.
`-nu+`	Add leading underscores to symbol names.
`-au-`	Remove leading underscores from public symbol names and keep old names as aliases.
`-au+`	Add leading underscores to public symbol names and keep old names as aliases.
`-nd`	Replace leading dot or underscore in nonstandard section names with the default for the target system.
`-nr:N1:N2`	Replace name `N1` with `N2`. `N1` may be a symbol name, section name or library member name.
`-ar:N1:N2`	Give public symbol `N1` an alias name `N2`. The same symbol will be accessible as `N1` as well as `N2`.
`-np:N1:N2`	Replace symbol prefix `N1` with `N2`. `N1` may be the beginning of a symbol name or section name.
`-nw:N1`	Make public symbol `N1` weak. Only possible for ELF files.

`-nl:N1`	Make public or external symbol `N1` local (invisible).
`-lx`	Extract all members from library `inputfile` to object files.
`-lx:N1:N2`	Extract member `N1` from library and save it as object file `N2`. The name of the object file will be `N1` if `N2` is omitted. May use `|` instead of `:` as separator.
`-la:N1:N2`	Add object file `N1` to library and give it member name `N2`. The member name will be `N1` if `N2` is omitted. May use `|` instead of `:`.
`-ld:N1`	Delete member `N1` from library.
`-v0`	Silent operation. No output to console other than warning and error messages.
`-v1`	Verbose. Output basic information about file names and types (Default).
`-v2`	More verbose. Tell about conversions and library operations.
`-wdXXX`	Disable warning number `XXX`.
`-weXXX`	Treat warning number `XXX` as an error.
`-edXXX`	Disable error message number `XXX`.
`-ewXXX`	Treat error number `XXX` as warning.
`-h`	Help. Print list of options.
`@RFILE`	Read additional command line parameters from response file `RFILE`.

Command line parameters can be stored in a response file. This can be useful if the command line is long and complicated. Just write `@` followed by the name of the response file. The contents of the response file will be inserted at the place of its name.

Response files can be nested, and there can be a maximum of ten response files.

Response files can have multiple lines and can contain comments. A comment starts with `#` or `//` and ends with a line break.

3 Warning and error control

Objconv can be called from a make utility. The make process will stop in case of an error message but not in case of warning messages. It is possible to disable specific error messages, to convert errors to warnings and to convert warnings to errors.

It is possible to disable error number 2005 is you want the input file and output file to have the same name. It is possible to disable error number 2505 if you want to mix object files with different word sizes in the same library.

4 Converting file formats

An object file can be converted from one format to another by specifying the desired format for the output file. The format of the input file is detected automatically. For example, to convert the 32-bit COFF file `file1.obj` to ELF:

```
objconv -felf32 file1.obj file1.o
```

The name of the output file will be generated, if it is not specified, by replacing the extension of the input file with the default extension for the target format. The name of the output file must be different from the input file.

The output file will always have the same word size as the input file. It is not possible to change the word size.

A library is converted in the same way as an object file:

```
objconv -felf32 file1.lib file1.a
```

You may use the `-nu` option to add or remove leading underscores on symbol names.

Debug information and exception handling information is removed from the file, by default, if the format of the output file is different from the input file. It is recommended to remove this information because it will be incompatible with the target system. Objconv does not include a facility for converting this information to make it compatible. You may get an unresolved reference named `__gxx_personality_v0` when converting from Gnu systems to another format if the exception handling information is not removed.

The input file should preferably be generated from assembly code with careful observation of the calling conventions of the target system. Compiler-generated code should not be converted but recompiled on the target platform. If the source code is not available then it may be necessary to disassemble the object file, fix any incompatibilities in the assembly code, and then assemble again. Linux, BSD and Mac systems are very similar and may be compatible with each other without recompiling.

The reasons why conversion of compiler-generated code to a different format may not work can be summarized as follows:

Reasons why conversion of compiler-generated code may fail	
Calling conventions in 64-bit mode	The calling conventions in 64-bit Windows and 64-bit Linux are very different. Functions with integer parameters will not work. Windows functions may use a shadow space not available in Linux. Linux functions may use a red zone not available in Windows.
Calling conventions in 32-bit mode	Most compilers have the same calling conventions in 32-bit mode, except for class member functions in Microsoft-compatible compilers. Use the keyword `__cdecl` on member function declarations in C++ on Microsoft-compatible compilers to force a compatible calling convention, or use `friend` functions instead of member functions.
Register usage conventions in 64-bit mode	Linux functions may modify registers `RSI`, `RDI` and `XMM6` - `XMM15`, which must be preserved by Windows functions.
Register usage conventions in 32-bit mode	The register usage conventions are the same in all 32-bit systems except for Watcom compilers.
Mangling of function names	Different compilers use different name mangling schemes. Use `extern "C"` on all function declarations in C++ to avoid name mangling. If this

	is not possible then you may have to change the mangled name by using the `-nr` option in objconv.
Leading under-scores on names	Use the `-nu` option on objconv to add or remove leading underscores when converting 32-bit files.
Initialization and termination code	Initialization and termination code is used for calling the constructors and destructors of global objects and for initializing function libraries. This code is not compatible between different systems.
Exception handling and stack unwinding information	This information is not compatible between different systems. Do not rely on structured exception handling.
Virtual tables and runtime type identification	Do not use virtual member functions or runtime type identification.
Communal functions and data	Objconv does not include a feature for converting communal (coalesced) data. Do not use function-level linking (`/Gy`) on Microsoft compilers or `-ffunction-sections` on Gnu compilers.
Incompatible relocation types	Mach-O files allow a relocation type that computes addresses relative to an arbitrary reference point. This is not supported by other systems. This is one of the reasons why objconv cannot convert from Mach-O to other file formats. 64-bit COFF files may contain image-relative relocations. This is not supported in other file formats.
Compiler-specific library calls	Most compilers can generate calls to library functions that are specific to that particular compiler. It may be necessary to convert the library function as well or make a replacement for this function.
Calls to operating system	Operating system calls are not compatible among systems.
Position-independent code	Linux, BSD and Mac systems require position-independent code when making shared objects (*.so). Windows systems are not able to make position-independent code. Use static linking when using converted object files on these systems. Avoid conversion of compiler-generated position-independent code. See `pic.asm` in `source.zip` for instructions on how to make position-independent code in MASM.
Lazy binding	Object files that use lazy binding cannot be converted because the import tables are not compatible with other systems.
Default library information	Information in object files about which libraries to include is not converted by objconv.

More details about incompatibilities between different platforms are documented in my manual number 5: "Calling conventions for different C++ compilers and operating systems". (www.agner.org/optimize).

My manual number 2: "Optimizing subroutines in assembly language" explains how to make function libraries that are compatible with multiple platforms. (www.agner.org/optimize).

5 Modifying symbols

It is possible to modify the names of public and external symbols in object files and libraries in order to prevent name clashes, to fix problems with different name mangling systems, etc.

Note that symbol names must be specified in the way they are represented in object files, possibly including underscores and name mangling information. Use the dump or disassembly feature to see the mangled symbol names.

To change the symbol name `name1` to `name2` in COFF file `file1.obj`:

```
objconv -fcoff -nr:name1:name2 file1.obj file2.obj
```

The modified object file will be `file2.obj`. Objconv will replace `name1` with `name2` wherever it occurs in public, external and local symbols, as well as section names and library member names. All names are case sensitive.

It is possible to give a public symbol more than one name. This can be useful for the purpose of supporting multiple name mangling schemes with the same object or library file. To give the function named `function1` the alias `function2`:

```
objconv -fcoff -na:function1:function2 file1.obj file2.obj
```

Some file formats have symbol names prefixed by an underscore (`_`) while other file formats have no prefix on symbol names. Use option `-nu` to change the prefix to the default for the target file format:

```
objconv -felf -nu file1.obj file2.o
```

You can specify any prefix to change or remove. For example, to remove prefix `_Win_` from all function names beginning with `_Win_`:

```
objconv -fcoff -np:_Win_: file1.obj file2.obj
```

No more than one operation can be specified for the same symbol name. For example, you cannot remove an underscore from a name and make an alias at the same time. You have to run objconv twice to so. For example, to convert COFF file `file1.obj` to ELF, remove underscores, and make an alias:

```
objconv -felf -nu file1.obj file1.o
objconv -felf -na:function1:function2 file1.o file2.o
```

Likewise, you have to run objconv twice to make two aliases to the same symbol.

It is possible to make a public symbol weak in ELF files. A weak symbol has lower priority so that it will not be used if another public symbol with the same name is defined elsewhere. This can be useful for preventing name clashes if there is a risk that the same function is supplied in more than one library. Note that only the ELF file format supports this feature. To make public symbol `function1` weak in ELF file `file1.o`:

```
objconv -felf -nw:function1 file1.o file2.o
```

COFF and OMF files have a different feature called weak external symbols. This is not supported by objconv.

Objconv can hide public symbols by making them local. A public symbol can be made local if you want to prevent name clashes or make sure that the symbol is never accessed by any other module. To hide symbol `DontUseMe` in COFF file `file1.obj`:

```
objconv -fcoff -nl:DontUseMe file1.obj file2.obj
```

It is also possible to hide external symbols. This can be used for preventing link errors with unresolved externals. The hidden external symbol will not be relocated. Note that it is dangerous to hide an external symbol unless you are certain that the symbol is never used. Any attempt to access the hidden symbol from a function in the same module will result in a serious runtime error.

All symbol modification options can be applied to libraries as well as to object files.

6 Managing libraries

A function library (archive) is a collection of object files. Each member (object file) in the library has a name which, by default, is the same as the name of the original object file. Objconv will always modify the member names, if necessary, to meet the following restrictions:

- Any path is removed from the member name.
- The member name (including extension) cannot be longer than 15 characters. This is for the sake of compatibility between different systems that differ in the way longer names are stored. The name is truncated if necessary.
- Member names must be different. Names that become identical after truncation will be modified to make them different.
- The member name must end in `.obj` for COFF and OMF files, or `.o` for ELF and Mach-O files. The extension is changed or added if necessary.

All libraries contain a symbol index in order to make it easier for linkers to find a particular symbol. Objconv will always remake the symbol index whenever a library file is modified.

`Objconv` can add, remove, replace, extract, modify or dump library members.

Rebuilding a library

To remove any path from member names and rebuild the symbol table in library `mylib.lib`:

```
objconv -fcoff mylib.lib mylib2.lib
```

Converting a library

To convert library `mylib.lib` from COFF to ELF format:

```
objconv -felf mylib.lib mylib.a
```

Building a library or adding members to a library

To add ELF object files `file1.o` and `file2.o` to library `mylib.a`:

```
objconv -felf -la:file1.o -la:file2.o mylib.a
```

or alternatively:

```
objconv -felf -lib mylib.a file1.o file2.o
```

The `-lib` syntax is intended for `make` utilities that produce a list of object files separated by spaces. The library `mylib.a` will be created if it doesn't exist.

If you want to preserve the original library without the additions then give the new library a different name:

```
objconv -felf -la:file1.o -la:file2.o mylib.a mylib2.a
```

Any members of the old library with the same names as the added object files will be replaced. Members with different names will be preserved in the library.

Any specified options for format conversion or symbol modification will be applied to the added members, but not to the old members of the library.

Removing members from a library

To delete member `file1.o` from library `mylib.a`:

```
objconv -felf -ld:file1.o mylib.a mylib2.a
```

Extracting members from a library

To extract object file `file1.o` from library `mylib.a`:

```
objconv -felf -lx:file1.o mylib.a
```

To extract all object files from library `mylib.a`:

```
objconv -felf -lx mylib.a
```

Any specified options for format conversion or symbol modification will be applied to the extracted members, but the library itself will be unchanged.

No more than one option can be specified for each library member. For example, you can't extract and delete the same member in one operation.

Modifying library members

To rename library member `file1.o` to `file2.o` in library `mylib.a`:

```
objconv -felf -nr:file1.o:file2.o mylib.a mylib2.a
```

To rename symbol `function1` to `function2` in library `mylib.a`:

```
objconv -felf -nr:function1.o:function2.o mylib.a mylib2.a
```

Any symbol modification option specified will be applied to all library members that have a symbol with the specified name.

Dumping library contents

To show all members and their public symbol names in library `mylib.a`:

```
objconv -fd mylib.a
```

Note that the member names shown are the names before conversion. All other commands use the member names after any path has been removed and the length has been limited to 15 characters.

To show the complete symbol list of member `file1.o` in library `mylib.a`:

```
objconv -fdhs -lx:file1.o mylib.a
```

7 Dumping files

Objconv can dump file headers, symbol tables, etc. for various types of files. For example, to dump the file header, section headers and symbol table of `file1.obj`:

```
objconv -fdfhs file1.obj
```

8 Disassembling files

Objconv can disassemble object files, executable files, etc. For example, to disassemble the dynamic link library `file1.dll`:

```
objconv -fasm file1.dll file1.asm
```

The output file uses standard MASM syntax. It is attempted to make the output file fully compatible with the Microsoft assembler (MASM or ML). However, it may be necessary to make minor modifications in the output file before it can be assembled, for example if symbol names contain dots or other characters that MASM syntax doesn't allow, if segments have a non-default alignment, or if the source file contains relocations for position-independent code that is not supported by MASM.

The disassembler supports the full instruction set for all 16, 32 and 64 bit x86 Intel and AMD processors, including the Intel SSE4 instruction set, AMD SSE5, privileged instructions, Intel VT instructions, and known undocumented instructions, totaling more than 1000 instructions.

The quality of the disassembly depends on the amount of information contained in the input file. Object files generally contain more information about symbol names, types, etc. than executable files do. COFF and ELF files contain more symbol names than OMF and Mach-O files do.

The disassembler goes to great lengths to distinguish between code and data, to determine the type of each data item, to guess where each function begins and ends, to identify import tables, jump tables, virtual function tables, etc. Nevertheless, the disassembler may sometimes misinterpret data as code, or fail to determine the type of a data item. When the disassembler is in doubt whether something is code or data, it will show it as both.

In simple cases, the quality of the disassembly may be good enough for making modifications in an object file or for extracting a single function from a dynamic link library. The disassembly of an executable file is unlikely to be good enough for remaking a fully working executable, but it is often good enough for identifying problems in the code.

8.1 How to interpret the disassembly

The following example shows what a piece of disassembled code may look like:

```
_text   SEGMENT PARA PUBLIC 'CODE'                      ; section number 1

?testb@@YAHH@Z PROC NEAR
        mov     eax, dword ptr [esp + 04H]              ; 0000 _ 8B. 44 24, 04
; Note: Memory operand is misaligned
        mov     ecx, dword ptr [?alpha@@3HA]            ; 0004 _ 8B. 0D, 00000000(d)
        add     ecx, eax                                ; 000A _ 03. C8
        push    ecx                                     ; 000C _ 51
        call    ?testa@@YAHH@Z                          ; 000D _ E8, 00000000(rel)
        add     esp, 4                                  ; 0012 _ 83. C4, 04
```

```
        mov     ecx, offset ?list1@@3PAHA               ; 0015 _ B9, 00000000(d)
; Filling space: 06H
; Filler type: lea with same source and destination
        db 8DH, 9BH, 00H, 00H, 00H, 00H
ALIGN   8
?_001:  add     eax, dword ptr [ecx]                    ; 0020 _ 03. 01
        add     ecx, 4                                  ; 0022 _ 83. C1, 04
        cmp     ecx, offset ?list1@@3PAHA + 00001000H   ; 0025 _ 81. F9, 00001000(d)
        jl      ?_001                                   ; 002B _ 7C, F3
        ret                                             ; 002D _ C3
?testb@@YAHH@Z ENDP
_text   ENDS
```

This code can be interpreted as follows:

The name `?testb@@YAHH@Z` is the name of the function `int testb(int x)` as it is mangled by the Microsoft C++ compiler. The disassembler does not translate mangled names to C++ names for you. The MASM assembler allows the characters `? @ $ _` in symbol names.

Line `0000` is the first instruction of the function `testb`. It reads the parameter `x` from the stack into register `eax`. Line `0004` reads a value from a variable in the data segment into `ecx`. The name `?alpha@@3HA` is a mangled name for `int alpha`. The note indicates that `alpha` is not optimally aligned. Such notes always apply to the instruction that follows. Line `000A` adds the value of `x` in `eax` to the value of `alpha` in `ecx`. Line `000C` pushes this value on the stack as a parameter to the following function call. Line `000D` is a call to function `int testa(int)` with a mangled name. The return value is in `eax`. Line `0012` cleans up the stack after the function call. Line `0015` loads the address of `?list1@@3PAHA` into `ecx`. This is the mangled name of an array `int list1[]`.

Next comes a multi-byte `nop` for aligning the subsequent loop entry. The compiler has used `lea ebx,[ebx+00000000H]` instead of 6 `nop` instructions for filling 6 bytes. The disassembler has written the exact byte sequence in order to avoid that the `lea` instruction is replaced by a shorter version when the code is re-assembled. The disassembler cannot know whether the desired alignment is 8 or 16. It is recommended that you remove the filler bytes and write `align 8` or `align 16` if you need to re-assemble the code. The assembler will then insert an appropriate multi-byte `nop`.

Line `0020` is a loop entry with the label `?_001`. The input file does not indicate a name for this label. Therefore the disassembler has assigned the arbitrary name `?_001`. Subsequent nameless code and data labels will be named `?_002`, etc.

The first line in the loop reads an integer from the address that `ecx` points to, i.e. an element from array `list1`, and adds it to `eax`. Line `0022` adds 4, which is the size of each array element, to `ecx` in order to make it point to the next array element.

Line `0025` compares `ecx` with the address of the end of the array. Line `002B` reads the flags from the preceding `cmp` instruction and jumps back to the top of the loop if the end of the array has not been reached. Line `002D` returns from function `testb`. The return value is in `eax`.

This code could be translated back to C++:

```
int testa(int x);
int list1[1024];
int alpha;

int testb(int x) {
   int y = testa(x + alpha);
   for (int i=0; i<1024; i++) y += list1[i];
```

```
    return y;
}
```

The comments to the right of the disassembly code are interpreted as follows. The four digits after the semicolon is the hexadecimal address of the instruction. This is actually a 32-bit value, but in this case the disassembler has saved some space by using only 4 hexadecimal digits. After the underscore comes the instruction code as hexadecimal bytes. The delimiters `: . ,` separate the different parts of the instruction code.

The text in parenthesis after the binary code indicates various types of cross-references, using the following abbreviations:

Abbreviation	Cross reference type
d	Direct address. The absolute virtual address of target is inserted
rel	Self-relative address
imgrel	Image-relative address
segrel	Address is relative to a segment or group
refpoint	Address is relative to an arbitrary reference point
seg	A segment address or segment descriptor
sseg	Only the segment part of a far target address is inserted
far	Offset and segment of a far target address
GOT	Global offset table entry
GOT r	Self-relative address of global offset table entry
PLT r	Self-relative address of procedure linkage table entry

The information about cross-reference types is usually obtained from relocation tables in the input file. The disassembler will attempt to reconstruct missing cross-reference information, if possible, in the case of executable files without relocation tables.

8.2 Checking the syntax of machine code

The disassembler adds notes to the output file if an instruction could be coded in a more efficient way or if there are syntax errors in the code. This can be useful for debugging purposes and for testing compilers and assemblers during development.

The type of syntax errors that the disassembler can detect are errors in an individual opcode, for example a memory operand on an instruction that allows only register operands. Objconv cannot detect errors in the programming logic, such as a `PUSH` that is not matched by a later `POP`.

Objconv will add notes in the disassembly for opcodes that could be coded in a more efficient way; for example an instruction that could be made smaller by replacing a 32-bit constant with a sign-extended 8-bit constant.

A note or error message does not necessarily indicate an error in the compiler that built the code. Compilers may sometimes have good reasons for coding an instruction in an apparently suboptimal form. Error messages typically occur when the compiler has placed data in the code segment and the disassembler has failed to identify this as data. It is very unlikely that the error messages you see are caused by bugs in the compiler.

9 Frequently asked questions

9.1 Why is there no graphical user interface?

Most users will prefer to call objconv from a make utility, a script or a batch file. A graphical user interface would compromise the cross-platform portability of the source code.

9.2 What kind of files can objconv convert?

Objconv can convert object files (*.obj, *.o) and static library files (*.lib, *.a) for 32-bit and 64-bit x86 systems, such as Windows, Linux, BSD and Intel-based Mac OS X.

The conversion is most likely to be successful if the file is built from assembly code with careful consideration of the calling conventions etc. of the target system. Conversion of compiler-generated code for 32-bit systems will work in simple cases where there are no system calls or other features known to cause problems. Conversion of 64-bit compiler-generated code will generally not work.

Se page 6 for a list of reasons why conversions may fail.

9.3 Can I build a function library that works in all operating systems?

Yes. If you build a static function library from an assembly language source code and you take care to obey all function calling conventions etc. then you can use objconv to convert the library to different file formats so that it works in all x86 systems. You need to make one version for 32-bit systems and another version for 64-bit systems. See my manual 2: "Optimizing subroutines in assembly language" for details.

9.4 Can I convert an executable file from Windows to Linux?

No. It is not possible to convert executable files between systems because they contain incompatible system calls. You may run the executable file under a Windows emulator (Wine).

9.5 Can I convert a dynamic link library to another system?

No. Objconv does not support the conversion of dynamic link libraries and shared objects.

9.6 Why can't I convert an export library?

The export library contains no function code. It only contains references to a DLL.

9.7 Can I convert a static library to a dynamic library?

Yes. You don't need objconv for this. The standard linker can do this. You only have to add a simple entry function.

9.8 Can I convert from a dynamic library to a static library?

No. If the source code is not available then you will have to disassemble the DLL and identify the function or functions you need. Then re-assemble this code. This is no easy job, but it may be possible in simple cases.

9.9 Can I convert from 32 bit code to 64 bit code?

No. The instruction codes are not compatible.

9.10 Can I convert a Windows function library to use it under other systems?

It may be possible to convert a 32-bit Windows function library (*.lib) and use it under Linux and other systems if the library contains no calls to system functions and no access to vendor-specific variables or functions. Conversion of commercial function libraries is unlikely to work. 64-bit Windows code is not compatible with 64-bit Linux.

9.11 I have problems porting my Windows application to Linux or Mac because the Gnu compiler has a strict syntax. Can I convert the compiled Windows code instead?

While you are trying to solve a small problem you are creating a much bigger problem. There are so many compatibility problems when converting compiler-generated code that this method is doomed to failure. Try to use a compiler that supports both operating systems, such as Intel or Gnu.

9.12 Is it possible to extract one or more functions from a binary file or program?

It is possible to extract single modules from a library file (`*.lib`, `*.a`), but it is not possible to automatically extract a function from an object file, executable file or dynamic link library. The file may contain spaghetti code that makes it impossible for the objconv program to tell where each function begins and ends. You may look at a disassembly to search for the function you need. If it is clear where the function begins and ends, and if the function is independent of other functions, then it may be possible to isolate this function and assemble it again.

9.13 Is it possible to convert mangled function names?

It is very tedious to do this manually. As yet there is no tool available for converting mangled names automatically. The Microsoft mangled names contain more information than the Gnu mangled names do, so it would be preferable to convert from Windows to Linux rather than vice versa. Se my manual 5: "Calling conventions for different C++ compilers and operating systems".

9.14 Is it possible to convert function calling conventions automatically?

No conversion is needed when converting between different 32-bit systems, except for class member functions using the Microsoft `__thiscall` convention and in rare cases differences in stack alignment. A conversion is needed when converting 64-bit object files because Windows and Linux systems use different calling conventions in 64-bit mode.

It is possible to make a call stub in assembly code that does the necessary conversions. The function call will then have to go through the call stub. You have to take care of all differences in parameter transfer conventions, register usage conventions, and stack shadow space. There may be a problem when converting from 64-bit Linux to 64-bit Windows if the function uses the red zone on the stack. Se my manual 5: "Calling conventions for different C++ compilers and operating systems".

It might be possible, at least in principle, to construct a tool that makes such a call stub automatically based on the information of function parameter types contained in the mangled function names. This would not work, however, for parameters of composite type because the mangled function names do not contain enough information to predict how a class object parameter is transferred. I am not going to build such a tool.

9.15 Does the disassembler have an interactive feature?

No. The current version of objconv has no feature for manually telling the disassembler what is code and what is data, etc.

9.16 Is it possible to disassemble an executable file to modify it and then assemble it again?

The disassembly of an executable program file is unlikely to contain enough information for reconstructing a fully working executable. It may be possible to do this on a DLL in simple cases.

9.17 Is it possible to disassemble an object file and fix all compatibility problems manually?

If you are an expert, yes. Many compatibility problems can be fixed manually. But this is hard work and there are many pitfalls. This is not for the faint-hearted!

9.18 Is it possible to reconstruct C++ code from a disassembly?

Reconstructing the logic behind a code from the disassembly is a lot of detective work, but it is possible with very small files. The disassembly of a program file typically contains hundreds of thousands of code lines. Interpreting so much code is simply an unmanageable job.

9.19 Why do I get error messages in the disassembly file?

Most disassembly errors occur because the compiler has placed data in the code segment and the disassembler attempts to interpret these data as code. The disassembler does its best to distinguish between code and data, but it is not always successful at this.

The disassembler will sometimes show the same binary data both as code and as data if it is in doubt what it is.

Data in the code segment should be avoided because this leads to inefficient caching and code prefetching. Unfortunately, some compilers are still putting jump tables etc. in the code segment. Older compilers do this a lot.

9.20 Can I disassemble byte code?

Objconv cannot convert or disassemble the byte code that is used for .net or Java. There may be other tools available for this.

9.21 I have problems assembling the output of the disassembler

Use the MASM assembler. `ml.exe` for 32-bit assembly and `ml64.exe` for 64 bit assembly.

Unfortunately, assembly language syntax is not very well-defined and not fully consistent. Statements that are allowed in 32-bit mode may not be allowed in 64-bit mode, etc. Therefore, you may need to make minor modifications in the disassembly output before it can be assembled again.

The assembler may give error messages if the alignment of the standard segments is different from the default. The default alignment for `_text` and `_data` is `para` (16) in 64 bit mode or if `.xmm` is specified, and `dword` (4) if `.xmm` is not specified. If the default alignment

doesn't fit your purpose then append a `$`-sign and something to the segment name, e.g. `_text$align32` and specify the desired alignment.

The name of all code segments must be `_text` or `_text$something` to make MASM recognize them as code segments.

Symbol names containing dots (`.`) must be changed to something else before assembling. MASM accepts symbol names beginning with a dot if `option dotname` is specified. MASM never accepts dots anywhere else in a symbol name because the dot is normally used as a structure member operator.

The assembler may not allow group directives if there is a `.model` directive. Remove lines like `FLAT GROUP` if they cause problems.

Debug segments may contain segment-relative references that MASM will not allow in a flat memory model. Remove the debug segments and any references to symbols in these segments.

MASM does not support communal data and functions. The disassembler will insert a note at communal sections and convert it to non-communal code.

9.22 Why does the disassembler use MASM syntax?

I have decided not to support the syntax of other assemblers such as GAS, NASM, YASM, FASM, WASM, etc. because I believe that we need a standardization of assembly syntax, and MASM is the closest we get to a *de facto* standard, despite its deficiencies.

The open source community ought to set up a working group for defining a standard for x86 assembly syntax and make tools that support it. This standard syntax should be a superset of MASM syntax in order to handle legacy code.

We cannot rely on any commercial company to maintain a good assembler because this is obviously not a profitable enterprise.

9.23 Why does my disassembly take so long time?

The handling of symbol tables etc. in objconv is not optimized for very large files. Converting or disassembling files of megabyte size can take several minutes. The handling of small to medium size files goes very fast.

9.24 How can I save the output of the dump screen to a file?

```
objconv -fdhs myfile > outputfile.txt
```

9.25 Can you help me with my programming problems?

No. I am not doing programming work for others, regardless of how much you pay. Sorry.

9.26 Are there any alternatives to objconv?

There are certain alternative tools that can convert and manipulate object files.

Intel's C++ compiler can compile the same source code on both Windows, Linux, BSD and Mac OS X platforms (www.intel.com). There are various versions of the Gnu C++ compiler for all platforms as well, although the Windows version is not as good as the versions for other platforms.

The Gnu `objcopy` utility can convert between various object file formats. The `objcopy` utility can be recompiled to support the file formats you need.

The Microsoft linker and library manager can convert from 32-bit OMF to COFF. The `Editbin` tool that comes with Microsoft compilers can convert from 32-bit OMF to COFF and modify COFF files.

The Digital Mars compiler includes a tool named `COFF2OMF` for converting 32-bit COFF files to OMF, and a disassembler `OBJ2ASM` that can disassemble object files in OMF, COFF and ELF format.

The Open Watcom compiler includes a disassembler called `WDISASM` and other utilities.

The `tdump` utility that comes with Borland compilers is useful for dumping COFF and OMF files, including executable files.

`debug.exe`. Comes with most versions of Windows. Can disassemble, debug and modify 16-bit executables.

10 Source code

The source code can be used for building the objconv executable for a particular platform and for modifying the program. The code is in C++ language and can be compiled with almost any modern C++ compiler that supports 64-bit integers on any platform with little-endian memory organization. The code has been tested with Microsoft, Intel and Gnu compilers. The code cannot run on platforms with big-endian memory organization, such as the PowerPC-based Mac.

You don't need to read the rest of this chapter unless you want to modify the source code of objconv.

10.1 Explanation of the objconv source code

The source code is intended to be compatible with all C++ compilers. Any modified code should preferably be tested on more than one compiler, including the Gnu compiler which has the strictest syntax checking.

Unfortunately, the C++ syntax has no standardized way of defining integers with a specific number of bits. Therefore, it is essential that you use the type definitions in `maindef.h` for defining integers with a specific size, e.g. `int32` for a 32-bit signed integer, and `uint32` for an unsigned 32-bit integer.

All dynamic data allocation must use the container classes declared in `containers.h` in order to prevent memory leaks. The following container classes are available:

`CMemoryBuffer` is useful for containing binary data of mixed type. You can append a data object `x` of any type to an instance `A` of `CMemoryBuffer` with `A.Push(&x,sizeof(x))`. You can append a zero-terminated ASCII string `s` with `A.PushString(s)`. You can read a data object `x` of type `mytype` stored in `A` at offset `os` with `x = A.Get<mytype>(os);` or `x = *(mytype*)(A.Buf() + os);` The former method does not work with old versions of the Gnu compiler if `A` is an instance of a template class derived from `CMemoryBuffer`, such as `CELF<>`. Use the type casting method in `CELF` and its descendants.

Note that it is dangerous to make a pointer to an object stored in a container because the internal buffer in the container class instance can be re-allocated when new data are added to the buffer. In some cases, the source code does use the unsafe technique of storing pointers to such data, but only when there is certainty that nothing is added to the container after the pointer has been assigned.

The container class `CFileBuffer` is derived from `CMemoryBuffer`. It adds methods for reading and writing files and for detecting the type of a file.

`CTextFileBuffer`, derived from `CFileBuffer`, is used for ASCII files.

The overloaded operators `>>` and `<<` are used for transferring ownership of a memory buffer from one container to another. It works with all descendants of `CFileBuffer`.

The template classes `CArrayBuf<RecordType>` and `CSList<RecordType>` are used for dynamic arrays where all members have the same type `RecordType`. Instances of these classes can be used as simple arrays with the index operator `[]`. `CArrayBuf` allows `RecordType` to have constructors and destructor, `CSList` does not. A dynamic array of type `CArrayBuf` has a size which cannot be changed after it has been set. A dynamic array of type `CSList` can be appended or resized at any time.

`CSList` is useful for sorted lists. `A.PushSort(x)` will insert object `x` in the list `A` in the right position so that the list is kept sorted at all times. `A.PushUnique(x)` does the same, but avoids duplicates. The sort criterion is determined by defining the operator `<` for `RecordType`.

All conversions of data files are done by a number of converter classes, which are all descendants of `CFileBuffer`. A file buffer can convert the data it contains by creating an object of the appropriate converter class, transferring ownership of its data buffer to the converter class object, letting the converter class do the conversion, and then taking back ownership of the converted data buffer, as shown in this example:

```
void CConverter::OMF2COF() {
   // Convert OMF to COFF file
   COMF2COF conv;                    // Make object for conversion
   *this >> conv;                    // Give it my buffer
   conv.ParseFile();                 // Parse file buffer
   if (err.Number()) return;         // Return if error
   conv.Convert();                   // Convert
   *this << conv;                    // Take back converted buffer
}
```

The operators `>>` and `<<` can transfer ownership of the contained data buffer because the classes `CConverter` and `COMF2COF` are both descendants of `CFileBuffer`.

The converter class `CELF` and its descendants are template classes with all the data structures of 32-bit or 64-bit ELF files as template parameters. This is because of the considerable difference between the data structures in 32-bit and 64-bit ELF files. The templates are instantiated explicitly in the bottom of `elf.cpp`.

The reading and interpretation of command line parameters is done by the class `CCommandLineInterpreter`, which has a single instance `cmd`. `cmd` is a global object so that it can be accessed from all parts of the program without being passed as a parameter.

Another global object is the error handler `err`, which is an instance of the class `CErrorReporter`. All error reporting is done with `err.submit(ErrorNumber)`. Exceptions are not used, for reasons of performance.

The Gnu compiler version 4 has a problem with inheritance from template classes because of an overly strict interpretation of the so-called two phase lookup rule. This problem is circumvented by putting `this->` in front of every access to members of an ancestor class in a class derived from a template class. For example, to access `CELF<>::NSections` from `CELF2COF<>` (which is derived from `CELF<>`), you have to write `this->NSections`. It is recommended to test that the code can be compiled with the Gnu compiler in order to catch these problems.

10.2 How to add support for new file formats

Define an id constant `FILETYPE_NEWTYPE` in `maindef.h` to identify the new file type. Add functionality in `CFileBuffer::GetFileType()` in `containers.cpp` for detecting this file type and its word size (16, 32 or 64 bits). Add a name for this file type to `FileFormatNames[]` in `containers.cpp`.

Define a class `CNewType` derived from `CFileBuffer` with member functions for parsing and dumping files of this type. The class declaration goes into `containers.h`. The definition goes into a new `.cpp` file named after the new type. Define converter classes for converting to and from the COFF or ELF type analogously to the existing converter classes in `converters.h`. Each converter class is derived from the class for the file type you convert from. Add member functions to `CConverter` for each converter class. Add case statements in `CConverter::Go()` in `main.cpp` for each possible conversion. A conversion may go through multiple steps if there is no converter class for direct conversion between the two types. You may also define a converter class for converting from NewType to itself in order to make it possible to modify symbol names in a file of type NewType without converting to one of the base types COFF or ELF and back again.

If the new file type contains x86 or x86-64 code then you may add a converter class for disassembling the new type. See below for the interface to the disassembler.

Note that the different object file formats differ in the way self-relative references are defined in relocation records. ELF and Mach-O files define self-relative references relative to the beginning of the relocation source field. COFF and OMF files define self-relative references relative to the end of the instruction needing the reference, as the x86 processors do. The difference between the two methods is equal to the length of the source field plus the length of any immediate operand in the instruction.

Objconv does not support file types with big endian memory organization.

10.3 How to add features to the disassembler

Only file types based on the x86 instruction set and its many extensions can be handled by the disassembler in objconv.

To add support for disassembling a new file type, you first have to make a converter class, as explained above. The converter class creates an instance of `CDisassembler` and uses the following member functions of `CDisassembler`: Use `CDisassembler::Init` for defining file type and possibly image base. Use `CDisassembler::AddSection` for defining each segment or section. Sections are numbered sequentially, starting at 1. Use `CDisassembler::AddSymbol` for defining local, public and external symbols. These can be numbered in random order, but numbers must be positive and limited. Use `CDisassembler::AddRelocation` for defining all cross-references and relocatable addresses. These can refer to symbol numbers. Use `CDisassembler::Go` to do the disassembly after all sections, symbols and relocations have been defined. Finally, take ownership of the disassembly file `CDisassembler::OutFile`.

You can add support for new instruction codes by adding entries to the opcode tables in `opcodes.cpp`. New Intel opcodes are likely to be 3-byte opcodes beginning with 0F 38 through 0F 3B. These are defined in tables `OpcodeMap2` through `OpcodeMap5`. New AMD opcodes are likely to begin with 0F 24, 0F 25, 0F 7A or 0F 7B defined in tables `OpcodeMap66` through `OpcodeMap69`.

The meaning of each field in the opcode table records is defined in the beginning of `disasm.h`.

Modifications to the functionality of the disassembler go into `disasm1.cpp`. Modifications to the way the disassembly output looks or support for alternative assembly syntaxes go into `disasm2.cpp`.

10.4 File list

Files in objconv.zip	
instructions.pdf	This file
objconv.exe	Executable for Windows
source.zip	Complete source code
Files in source.zip	
build.sh	Script for building objconv for Linux, BSD and Mac systems
objconv.vcproj	Project file for Microsoft compiler
objconv.suo	Options file for Microsoft compiler
pic.asm	Example of how to make position-independent code in MASM
cmdline.cpp	Defines class CCommandLineInterpreter for reading command line
cof2asm.cpp	Defines class CCOF2ASM for disassembling COFF files
cof2cof.cpp	Defines class CCOF2COF for modifying COFF files
cof2elf.cpp	Defines class CCOF2ELF for converting from COFF to ELF
cof2omf.cpp	Defines class CCOF2OMF for converting from COFF to OMF
coff.cpp	Defines class CCOFF for parsing and dumping COFF files
containers.cpp	Container classes CMemoryBuffer, CFileBuffer, CTextFileBuffer
disasm1.cpp	Defines part of class CDisassembler for disassembling
disasm2.cpp	Defines part of class CDisassembler for disassembling
elf.cpp	Template class CELF for dumping and parsing ELF files
elf2asm.cpp	Template class CELF2ASM for disassembling ELF files
elf2cof.cpp	Template class CELF2COF for converting from ELF to COFF
elf2elf.cpp	Template class CELF2ELF for modifying ELF files
elf2mac.cpp	Template class CELF2MAC for converting from ELF to Mach-O
error.cpp	Defines class CErrorReporter and error texts
library.cpp	Defines class CLibrary for building and modifying .lib and .a files
mac2asm.cpp	Defines class CMAC2ASM for disassembling Mach-O files
macho.cpp	Defines class CMACHO for parsing and dumping Mach-O files
main.cpp	Classes CMain and CConverter for dispatching command
omf.cpp	Defines class COMF for parsing and dumping OMF files
omf2asm.cpp	Defines class COMF2ASM for disassembling OMF files
omf2cof.cpp	Defines class COMF2COF for converting from OMF to COFF
omfhash.cpp	Defines class COMFHashTable for hash tables in OMF libraries
opcodes.cpp	Tables for complete set of opcodes for disassembler
stdafx.cpp	Needed only for precompiled headers
cmdline.h	Declares class CCommandLineInterpreter and various constants
coff.h	Structures and constants for COFF files
containers.h	Declares container classes and container class templates
converters.h	Declares many converter classes derived from CFileBuffer

disasm.h	Declares several structures and classes used by disassembler
elf.h	Structures and constants for ELF files
error.h	Declares class CErrorReporter for error handling
library.h	Structures and classes for managing .lib and .a files
macho.h	Structures and constants for Mach-O files
maindef.h	Type definitions and other main definitions
omf.h	Structures, classes and constants for OMF files
stdafx.h	Includes all the other .h files

10.5 Class list

The most important container classes and converter classes in the objconv source code are listed below.

Container classes	
CMemoryBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** none **Description:** This is the base container class that all file classes, converter classes and all classes containing data of mixed types are derived from. The size can grow as new data are added.
CFileBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** CMemoryBuffer **Description:** This is the container class that all converter classes and other file handling classes are derived from. It adds methods for reading and writing files and for detecting the input file type.
CTextFileBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** CFileBuffer **Description:** Container class for reading and writing ASCII text files.
CArrayBuf<>	**Declared in:** containers.h **Defined in:** containers.h **Inherit from:** none **Description:** Container class template for arrays where all records have the same type. The record type is defined as a template parameter. The size cannot be modified after it has been set. The record type can have constructors and destructor.
CSList<>	**Declared in:** containers.h **Defined in:** containers.h **Inherit from:** CMemoryBuffer **Description:** Container class template for arrays where all records have the same type. The record type is defined as a template parameter. The size can grow as new records are added. The list can be sorted. The record type can not have constructors or destructor.

Classes for converting files, etc.	
CMain	**Declared in:** converters.h **Defined in:** main.cpp **Inherit from:** CFileBuffer **Description:** Dispatching input file to CConverter or CLibrary
CConverter	**Declared in:** converters.h **Defined in:** main.cpp **Inherit from:** CFileBuffer **Description:** Dispatching input file to any of the converter classes
CLibrary	**Declared in:** library.h

	Defined in: library.cpp **Inherit from:** CFileBuffer **Description:** Reading and building library files of any type
COMFHashTable	**Declared in:** library.h **Defined in:** omfhash.cpp **Inherit from:** none **Description:** Reading and building hash table for OMF libraries
CCOF	**Declared in:** converters.h **Defined in:** coff.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of COFF and PE files
CCOF2ELF	**Declared in:** converters.h **Defined in:** cof2elf.cpp **Inherit from:** CCOFF **Description:** Conversion from COFF to ELF
CCOF2OMF	**Declared in:** converters.h **Defined in:** cof2omf.cpp **Inherit from:** CCOFF **Description:** Conversion from COFF to OMF
CCOF2ASM	**Declared in:** converters.h **Defined in:** cof2asm.cpp **Inherit from:** CCOFF **Description:** Disassembly of COFF and PE files
CCOF2COF	**Declared in:** converters.h **Defined in:** cof2cof.cpp **Inherit from:** CCOFF **Description:** Modification of COFF files
COMF	**Declared in:** converters.h **Defined in:** omf.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of OMF files
COMF2COF	**Declared in:** converters.h **Defined in:** omf2cof.cpp **Inherit from:** COMF **Description:** Conversion from OMF to COFF
COMF2ASM	**Declared in:** converters.h **Defined in:** omf2asm.cpp **Inherit from:** COMF **Description:** Disassembly of OMF files
CELF<>	**Declared in:** converters.h **Defined in:** elf.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2COF<>	**Declared in:** converters.h **Defined in:** elf2cof.cpp **Inherit from:** CELF<> **Description:** Conversion from ELF to COFF. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2MAC<>	**Declared in:** converters.h **Defined in:** elf2mac.cpp **Inherit from:** CELF<> **Description:** Conversion from ELF to Mach-O. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2ASM<>	**Declared in:** converters.h **Defined in:** elf2asm.cpp **Inherit from:** CELF<>

	Description: Disassembly of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2ELF<>	**Declared in:** converters.h **Defined in:** elf2elf.cpp **Inherit from:** CELF<> **Description:** Modifications of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CMACHO	**Declared in:** converters.h **Defined in:** macho.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of Mach-O files
CMACUNIV	**Declared in:** converters.h **Defined in:** macho.cpp **Inherit from:** CFileBuffer **Description:** Parsing Mac universal binary files
CMAC2ASM	**Declared in:** converters.h **Defined in:** mac2asm.cpp **Inherit from:** CMACHO **Description:** Disassembly of Mach-O files
CDisassembler	**Declared in:** disasm.h **Defined in:** disasm1.cpp, disasm2.cpp, opcodes.cpp **Inherit from:** none **Description:** Disassembling code. Called from CCOF2ASM, COMF2ASM, CELF2ASM, CMAC2ASM
CSymbolTable	**Declared in:** disasm.h **Defined in:** disasm1.cpp **Inherit from:** none **Description:** Manage symbol table during disassembly.
CCommandLineInterpreter	**Declared in:** cmdline.h **Defined in:** cmdline.cpp **Inherit from:** none **Description:** Interpretation of command line parameters
CResponseFileBuffer	**Declared in:** converters.h **Defined in:** cmdline.cpp **Inherit from:** CFileBuffer **Description:** Contains response file from command line

11 Legal notice

Objconv is an open source program published under the conditions of the GNU General Public License, as defined in www.gnu.org/copyleft/gpl.html. The program is provided without any warranty or support.

It may not be legal to modify, convert or disassemble copyright protected software files without permission from the copyright owner. It is an open question whether it is legal to modify or convert a copyright protected function library and use it for other purposes or on other platforms than presupposed in the license conditions. It is recommended to ask the vendor for permission before developing and publishing any software that is built with the use of a converted copyright protected function library.

Copyright law does not generally permit disassembly of copyright protected software for the purpose of modifying the software, for circumventing a copy protection mechanism, for using part of the code in other contexts, or for extracting the algorithms behind the code.

European and US copyright law may, under certain conditions, permit reverse engineering of copyright protected software when the sole purpose is to extract the information

necessary for establishing interoperability with other software, e.g. to make other software capable of producing data files that are compatible with said copyright protected software, and only to the extent necessary for this purpose. However, I am not a legal expert. The user must seek legal advise before deciding whether it is legal to use objconv for any such purpose.