Cmajor Compiler User's Manual

Seppo Laakko

September 28, 2014

1 File Types

Table 1 shows the file types recognized by the Cmajor compiler.

Table 1: Cmajor Files

File Extension	Description
.cm	Cmajor Source File
.cmp	Cmajor Project File
.cms	Cmajor Solution File
.ll	LLVM Intermediate Code File
.opt.ll	Optimized LLVM Intermediate Code File
.c	C Intermediate Code File
.cdi	C Debug Information File
.cmdb	Cmajor Debugger Project Debug Information File
.o	Object Code File
.cml	Cmajor Library Interface File
.cma	Cmajor Assembly File
.dep	Dependency Information File
.exc	Exception Type File
.fun	Temporary Function File
.cmp.sym	Conditional Compilation Symbols
.cmp.usr	Project Property File

.cml file is similar to a C++ header file, but automatically generated by the compiler from the source files and for the whole library.
.cma file is actually an archive containing object code and generated by the `ar` command.
.trm file gives a mapping from a template specialization to the name of the file that contains it.

2 Compile Process

1. If the file argument for the compiler is a solution file (.cms), the compiler parses the solution file and does a topological sort for the projects the solution contains to determine the project build order.

   If the file argument for the compiler is a project file (.cmp), the project build order consists trivially of the sole project file.

2. For each project in turn, the libraries (.cml files) the project uses are parsed.

3. The libraries the project uses are loaded.

4. The source files of the project are parsed and syntax trees for the objects in each source file are created. Each object is placed into its enclosing scope.

5. The signature of each object is type checked.

   The type checking is done in two phases [1].

   (a) The names contained in each class are resolved.

   (b) The signature of each free function and member function of a class is type checked. This type checking resolves the type of each function parameter, member variable and typedef definition.

   For each class the compiler generates a default constructor, copy constructor, copy assignment, destructor and static constructor when applicable. These generated members are type checked and function calls they generate are enqueued to the next stage.

6. The enqueued function templates and member functions of a class template are instantiated and intermediate code for them is generated.

7. Each source file in turn is compiled. The compilation consists of following phases.

   (a) The body of each free function and member function in a source file is type checked. The function templates the body calls are instantiated and compiled along the way recursively. The called function template specializations and member functions of class templates are instantiated and the intermediate code for them is generated to a .fun file.

   (b) Intermediate code for each object the source file contains is generated to a .ll file. Intermediate code for each called inline function is generated also to the .ll file that contains the calling function.

   (c) Object code for the .ll file is generated to an .o file using the LLVM static compiler (`llc`).

   If -emit-opt option is used, an optimized intermediate code file .opt.ll from the .ll file is generated using the LLVM optimizer (`opt`).

   If -S option is used, a native assembly source code file .s is generated from the .ll file using the LLVM static compiler (`llc`).

   (d) The memory for bodies of the functions in the compilation unit is released.

8. If compiling a program a compilation unit containing the main function (`__main__.ll` file) is generated compiled to object code (`__main__.o`) using `llc`. Also a compilation unit containing an exception table (`__exception_table__.ll` file) is generated and compiled to object code (`__exception_table__.o`) using `llc`.

9. If the project contains .c files they are compiled using `gcc`.

[1] This is done like this to prevent mutually dependent classes to cause problems.

10. A library file (.cml) for the project is generated.
11. Memory is released.
12. An archive (.cma) containing object code for the project is created using `ar`.
13. If compiling a program, archive files (.cma files) for the project and the libraries it uses are linked using `gcc` to form an executable file.

3 Compiler Output

When the compiler parses a source file for a template or concept definition it outputs a line of the form

```
PT> C:/Programming/cmajor++/system/eh.cm
```

When the compiler parses a source file for inline definition it outputs a line of the form:

```
PIN> C:/Programming/cmajor++/system/charclass.cm
```

4 Compile Options

```
usage: cmc [options] {file.cms | file.cmp}
build solution file.cms or project file.cmp
```

Compile options are shown in table 2.

5 Debugging Memory Leaks

If a program and all the libraries it uses (including the system library) are compiled with conditional compilation symbol MEM_LEAK_DETECT enabled [2], detected memory leaks are reported at the end of the program:

```
DBGHEAP: memory leaks detected...
serial=0, mem=00000000007512B0, size=4, file=C:/Programming/cmajor++/test/leak/leak.cm,
line=3
```

Then, if the serial number of the leak is given to `dbgheap_stop(int serial)` function as argument, the program stops at allocation number `serial`, prints the call stack and exits.

For example, the program...

```
void main()
{
    dbgheap_stop(0);
    int* p = new int(1);
}
```

[2] -D option

...prints the following information about leak number 0:

```
DBGHEAP> stop: serial=0, file=C:/Programming/cmajor++/test/leak/leak.cm, line=4,
call stack:
0> function 'main()' file C:/Programming/cmajor++/test/leak/leak.cm line 2
```

6 Compiler Debugging Options

6.1 -debug:cc

Debug compiling classes.

Generates output lines of the form `C> CLASS`, when the compiler starts to compile a class CLASS and `C< CLASS`, when the compiler has compiled a class CLASS.

6.2 -debug:cf

Debug compiling functions.

Generates output lines of the form `F> FUNCTION`, when the compiler starts to compile a free function FUNCTION and `F< FUNCTION`, when the compiler has compiled function FUNCTION.

Generates output lines of the form `M> MEMFUN`, when the compiler starts to compile a member function MEMFUN and `M< MEMFUN`, when the compiler has compiled a member function MEMFUN.

6.3 -debug:ti

Debug template instantiation.

Generates output lines of the form `T> TEMPLATE`, when the compiler starts instantiating a class or function template TEMPLATE and `T< TEMPLATE`, when the compiler instantiated a class or function template TEMPLATE.

6.4 -debug:arch

Debug archetypes.

Generates output lines of the form

```
A> TEMPLATE
=======================
System.Concepts.Archetype0_I
System.Concepts.Archetype0_T
=======================
```

listing each generated archetype.

If used with the `-verbose` option generates output lines of the form:

```
A> TEMPLATE
================================================================================
    ARCHETYPE
    {
```

```
        #member functions:
        ...
        #non-member functions:
        ...
        #associated types:
        ...
        #equivalence classes:
        ...
    }
```

6.5 -debug:aor

Debug ambiguous overload resolution.

Prints output lines that list overload candidates and required argument conversions.

6.6 -debug:sfp

Debug source file parsing.

Generates debug output when parsing source file `<file.cm>` into `<file>.parse.xml`.

This generates lots of output, so you may want to use it only for single source file compiles and only for small files.

6.7 -debug:mem

Debug memory usage.

Generates debug output containing sizes of various caches and tables. Gives only magnitudes, not exact measures.

Table 2: Compile Options

Option	IDE command	Meaning
-R	Build/Rebuild Solution	Rebuild project or solution for the selected configuration
-clean	Build/Clean Solution	Clean project or solution for the selected configuration
-config=<config>	Configuration combo box	Use <config> configuration. (Default is debug).
-backend=llvm	Backend combo box	Use LLVM backend (default).
-backend=c	Backend combo box	Use C backend.
-O=<n>		Set optimization level to <n> (0-3). Defaults: debug: O=0, release: O=3.
-c <file.cm>	right click source file/Compile	Compile single source file. Do not link.
-g	Build/Options/Generate debug information	Generate debug information.
-D <symbol>	right click project/Properties	Define conditional compilation symbol <symbol>.
-L <dir1>;<dir2>;...		Add directories <dir1>, <dir2>, ... to library directories.
-S	Build/Options/Generate native assembly source code	Generate native assembly source code to .s file.
-emit-opt	Build/Options/Emit optimized intermediate code file	Write optimized intermediate code to .opt.ll file.
-nocttc	Build/Options/No constrained template type checking	Do not type check constrained template using concepts. Speeds up the compile at the cost of reduced safety.
-time		Report compilation time for a successful compile.
-quiet	Build/Options/Keep quiet	Output only errors.
-verbose	Build/Options/Generate verbose output	Generate verbose output.
-stats	Build/Options/Print statistics	Report statistics at the end of a successful compile.