Cmajor Compiler User's Manual

Seppo Laakko

August 5, 2016

1 File Types

Table 1 shows the file types recognized or generated 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 Debug Information File
.o	Object Code File
.cml	Cmajor Library File
.cma	Cmajor Assembly File
.dep	Dependency Information File
.exc	Exception Type File
.bcu	Bound Compile Unit
.fun	Temporary Function Intermediate Code File
.cmp.sym	Conditional Compilation Symbols
.cmp.usr	Project Property File
.cms.usr	Solution Property File

.cma file is actually an archive containing object code and generated by the `ar` command.

2 Compile Process

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.

The following steps are executed for each project the solution contains:

1. The source files of the project are parsed and abstract syntax trees for the objects in each source file are created.
2. Project's symbol table is initialized.
3. The symbol tables of the libraries the project uses are imported to the project's symbol table.
4. The abstract syntax trees are traversed and symbols for AST nodes are created and inserted to the symbol table.

   AST nodes that have corresponding symbols are:

   - namespaces,
   - classes,
   - functions,
   - enumerated types and
   - enumeration constants,
   - typedefs,
   - delegates,
   - class delegates,
   - template parameters,
   - parameters,
   - variables,
   - code blocks (compound statements), for statetements and range for statements,
   - concepts.

5. C files the project contains are compiled using gcc.
6. C++ files the project contains are compiled using g++.
7. LLVM intermediate code files the project contains are compiled using llc.
8. Main compilation begins. The compilation consists of following phases.

   - Syntax trees for all compile units are traversed using the prebinder. The prebinder gets the symbol for the corresponding AST node and processes it. The prebinder checks that the specifiers for the symbol make sense. It also resolves type nodes to the corresponding type symbols.

     The nodes and symbols processed are: namespace imports and using aliases, classes, functions, variables, enumerated types and enumeration constants, constants, parameters, delegates and class delegates.
   - Syntax trees for all compile units are traversed using the virtual binder. The main task of the virtual binder is to initialize virtual function tables for class symbols. It also generates a destructor for a class if the class is virtual or has a nontrivial destructor for some member variable, or has base class that has a destructor. Finally the virtual binder generates a static constructor for a class if the the does not have a user defined static constructor but has a static member variable.

- Each compile unit in turn is bound using the main binder, intermediate code for it is generated using the emitter and the intermediate code is compiled to object code using llc (if LLVM backend is used) or gcc (if C backend is used).

  The main binder traverses a syntax tree for the compile unit and generates a corresponding bound tree. The bound tree contains nodes for classes, functions, statements and expressions. Each bound expression node has a resolved type symbol and overload resolution is used to resolve function calls to function symbols.

  The overload resolution also fires generation of syntax trees for function template specializations and for member functions of class template specializations. Generated syntax trees are recursively processed using the prebinder, virtual binder and main binder.

  The overload resolution also fires generation of synthesized class member functions. The synthesized class member functions are default constructors, copy constructors, copy assignments, move constructors and move assignments. They are automatically generated by the compiler if neeeded (called) unless suppressed by the user.

- The emitter processes the bound tree representation generated by the main binder and generates an intermediate code file (.ll for LLVM and .c for C backend). The intermediate code file is a text file that contains primitive instructions for each bound class, function, statement and expression. (See for example http://llvm.org/docs/LangRef.html.) The generated C code is very primitive and looks more like assembler.

- The generated intermediate code is feeded to static LLVM compiler (llc) or C compiler (gcc), that generate object code (.o file) for it.

9. If compiling a program, a compile unit containing main function is generated (`__main__.ll` or `__main__.c`). The main function calls user supplied main function that is renamed to user main. Also a compile unit containing an exception table is generated (`__exception_table__.ll` or `__exception_table__.c`).

10. The compiled object code files are feeded to archiver (ar) that creates an object code library (named .cma).

11. If compiling a program , the object code libraries are linked to an executable using gcc.

12. A library file (.cml) for the project is generated. The library file contains project's symbol table and abstract syntax trees for templates in binary form.

13. If compiling a program using C backend and debug configuration, a debug information file (.cmdb) is created.

3 Compile Options

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

Compile options are shown in table 2.

4 Target Triples and Datalayouts

Current version of the Cmajor compiler (1.2.0) emits LLVM target triples and datalayouts to .ll files to enable LLVM compiler to better optimize the code.

The LLVM target triple to use can be given using compiler option (-m) or an environment variable (CM_TARGET_TRIPLE). If no target triple is given the Cmajor compiler emits the default target triple for the given system unless -emit-no-triple compiler option is given.

The LLVM datalayout to use can be given using compiler option (-d) or an environment variable (CM_TARGET_DATALAYOUT). If no datalayout is given the Cmajor compiler emits the default datalayout for the given system unless -emit-no-layout compiler option is given.

The default target triples and datalayouts are:

Platform	**Target triple**	**Datalayout**
64-bit Windows	x86_64-w64-windows-gnu	e-m:e-i64:64-f80:128-n8:16:32:64-S128
32-bit Windows	i686-pc-windows-gnu	e-m:w-p:32:32-i64:64-f80:32-n8:16:32-S32
64-bit Linux PC	x86_64-pc-linux-gnu	e-m:e-i64:64-f80:128-n8:16:32:64-S128
32-bit Linux PC	i686-pc-linux-gnu	e-m:w-p:32:32-i64:64-f80:32-n8:16:32-S32

To determine what is the correct LLVM target triple and datalayout for other platforms you must obtain *clang*:

Clang for Windows (Cmajor uses mingw-version) can be obtained from:
http://sourceforge.net/projects/clangonwin/files/MingwBuild/3.6/llvm-win64-r222867-x86_64-w64-mingw32.zip/download (64-bit) or from:
http://sourceforge.net/projects/clangonwin/files/MingwBuild/3.6/LLVM-3.6.0svn-r218657-win3exe/download (32-bit).

Clang for Unix-like systems can be obtained from http://llvm.org/releases/download.html#3.7.0.

For determining the LLVM target triple and datalayout, create a minimal C program **target.c** and compile it using clang:

- **target.c**:

```
int main()
{
    return 0;
}
```

- run clang:

```
clang -S -emit-llvm -c target.c -o target.ll
```

- Inspect **target.ll**:

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-w64-windows-gnu"
```

- Set environment variables for Cmajor:

```
set CM_TARGET_TRIPLE=x86_64-w64-windows-gnu
set CM_TARGET_DATALAYOUT=e-m:e-i64:64-f80:128-n8:16:32:64-S128
```

To make the environment variables persistent edit environment variables in the Advanced System Settings pane in the System Control Panel (Windows) or edit your shell script (Unix-like systems).

5 Configurations

5.1 debug | LLVM Configuration

The default configuration is the **debug | LLVM** configuration. In this configuration `#assert` statements are in effect. Currently programs compiled using this configuration cannot be debugged. No optimizations are performed for this configuration.

5.2 debug | C Configuration

This configuration is selected when compiler option -backend=c is given. In this configuration `#assert` statements are in effect. Programs compiled using this configuration can be debugged from IDE in Windows or using command line tool cmdb. No optimizations are performed for this configuration.

5.3 release | LLVM Configuration

This configuration is selected when compiler option -config=release is given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled.

5.4 release | C Configuration

This configuration is selected when compiler options -config=release and -backend=c are given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled.

5.5 profile | LLVM Configuration

This configuration is selected when compiler option -config=profile is given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled. This configuration is used for profiling (see *Profiling Cmajor Applications*).

5.6 profile | C Configuration

This configuration is selected when compiler options -config=release and -backend=c are given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled. This configuration is used for profiling. Every function is instrumented by calls to get timestamps for start and end of the function. This configuration is used for profiling (see *Profiling Cmajor Applications*).

5.7 full | LLVM Configuration

This configuration is selected when compile option -config=full is given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled. This is an experimental configuration.

In addition to optimizations done in **release** configuration a whole-program analysis is performed. In the whole-program analysis the compiler builds a type propagation graph and uses it to devirtualize virtual function calls (see http://web.cs.ucla.edu/~palsberg/tba/papers/sundaresan-et-al-oopsla00.pdf). The original variable-type analysis is for Java, but I adapted it for Cmajor. In addition every class of the program and in the library the program uses is given a prime number (key). The keys are used to compute a unique class identifier for each class. The class identifier is the product of the key of the class and keys of each of its base class. The class identifiers computed this way enable to optimize **is** and **as** expressions. Normally **is** and **as** expression must traverse the class hierarchy, but in 'full' configuration determining if a class derives from another reduces to a simple modulo operation (see http://www.stroustrup.com/fast_dynamic_casting.pdf by Michael Gibbs and Bjarne Stroustrup.)

Due to whole-program analysis compiles take considerably longer than in other configurations and the compile process is different: First each compile unit is processed the same way as in other configurations. But instead emitting low-level intermediate code using emitter, the high-level intermediate representation <unitname>.bcu (.bcu stands for Bound Compile Unit) is saved. If compiling a library the compilation stops here. When compiling a program these .bcu files contained in the program and all the libraries it uses are read in turn and type propagation graph is built. Then the type propagation graph is processed (variable-type analysis) and class identifiers are computed for each virtual class. Finally each .bcu file is read again, low-level intermediate code is generated (.ll or .c file) and compiled to object code using LLVM compiler or C compiler. Then object files are feeded to archiver (ar). Finally that archive file (.cma) is linked using **gcc** to form an executable.

5.8 full | C Configuration

This configuration is selected when compiler options -config=full and -backend=c are given. In this configuration `#assert` statements have no effect. Programs compiled using this configuration cannot be debugged. Default optimization level for this configuration is O3. Function inlining is enabled. This is an experimental configuration.

The same whole-program analysis and optimizations it makes possible as in the **full | LLVM** configuration are performed.

6 Code Completion

Code completion in the editor is activated by pressing CTRL-K. It shows a list of possible symbol names along with their type tags in the current parsing context. Typing letters filters the symbol name list. You can scroll the names using the arrow and page up and page down keys. ENTER selects a symbol name. ESC or CTRL-K hides the code completion list. After selecting a symbol name from the list a yellow code completion item window is shown. You can scroll through the possible code completion items using the arrow keys. Pressing

ENTER, ESC, CTRL-K or semicolon hides the code completion item window. The following table shows the possible symbol type tags:

Tag	Symbol
F	Function
C	Constant
E	Enumeration Constant
V	Variable
T	Type
N	Namespace
O	Concept

7 Interoperability with C and C++

To call a C function **foo(int)** from a Cmajor program, you must declare it in some Cmajor source file as **extern public cdecl void foo(int arg);**. If the **foo** function is contained in the C library **foolib.lib** you can include the library by declaring it in the Cmajor project file as **clib <path/to/foolib>;**. If you have a C source file **foo.c** that contains the **foo** function you can include it in the Cmajor project file as **csource <foo.c>;**.

To call a C++ function **foo(int)** in **foo.cpp** from a Cmajor program the easest way is to define it as `extern "C" foo(int arg) { ... }` in **foo.cpp** and declare it in some Cmajor source file as **extern public cdecl void foo(int arg);**. Then you can include it in the Cmajor project file as **cppsource <foo.cpp>;**. If the **foo** function is contained in a library and it is declared as `extern "C"` you can proceed the same as in the C case. However if the **foo** function is contained in a library and it is not declared as `extern "C"` you must obtain the mangled name for it somehow (for example **fooi3lx**) and declare it in some Cmajor source file as **extern public cdecl void fooi3lx(int arg);**. Then you can include the library by declaring it in the Cmajor project file as **clib <path/to/foolib>;**.

To call a Cmajor function **foo(int)** from a C program define the function as `cdecl void foo(int arg) { ... }` in **foo.cm**. Then you can declare it as **void foo(int);** in some **foo.h** and provide the **.cma** file generated by the Cmajor compiler to the C compiler.

To call a Cmajor function **foo(int)** from a C++ program define the function as `cdecl void foo(int arg) { ... }` in **foo.cm**. Then you can declare it as **extern "C" void foo(int);** in some **foo.h** and provide the **.cma** file generated by the Cmajor compiler to the C++ compiler.

8 Tracing

If a program and all libraries it uses are compiled with -trace compiler option enabled, execution trace is printed. For example, for the following program...

```
using System;

int foo(int x)
{
    return 1;
}
```

```
void bar(double x)
{
    foo(0);
}

void main()
{
    bar(2.0);
}
```

... the following trace is printed:

```
>0000:main()[C:/Programming/cmajorbin/test/tracing/tracing.cm:14]
+0001:main()[C:/Programming/cmajorbin/test/tracing/tracing.cm:15]
>0002:bar(double)[C:/Programming/cmajorbin/test/tracing/tracing.cm:9]
+0003:bar(double)[C:/Programming/cmajorbin/test/tracing/tracing.cm:10]
>0004:foo(int)[C:/Programming/cmajorbin/test/tracing/tracing.cm:4]
<0004:foo(int)[C:/Programming/cmajorbin/test/tracing/tracing.cm:4]
-0003:bar(double)[C:/Programming/cmajorbin/test/tracing/tracing.cm:10]
<0002:bar(double)[C:/Programming/cmajorbin/test/tracing/tracing.cm:9]
-0001:main()[C:/Programming/cmajorbin/test/tracing/tracing.cm:15]
<0000:main()[C:/Programming/cmajorbin/test/tracing/tracing.cm:14]
```

9 Debugging Memory Leaks

If a program and all libraries it uses are compiled with -debug_heap compiler option enabled, a report of the detected memory leaks is printed at the end of the program:

```
DBGHEAP> memory leaks detected...
serial=0, mem=0000000000701A50, size=4
```

Then, if the serial number of the leak is given as argument to the `dbgheap_watch(int serial)` function, The program prints call stack when allocation of that serial number occurs.

For example, program:

```
void leak()
{
    int* x = new int(1);
}

void main()
{
    dbgheap_watch(0);
    leak();
}
```

Prints the following call stack:

```
call stack:
1> function 'System.Support.DebugHeapMemAlloc(ulong)' file C:/Users/Seppo/AppDat
a/Roaming/Cmajor/system/utility.cm line 136
0> function 'main()' file C:/Programming/cmajorbin/test/leak_test/leak_test.cm
line 9
```

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
-c `<file.cm>`	right click source file | Compile	Compile single source file. Do not link.
-config=`<config>`	Configuration combo box	Use `<config>` configuration. (Default is `debug`).
-D SYMBOL	Edit project properties	Define conditional compilation symbol SYMBOL.
-O=`<n>`		Set optimization level to `<n>` (0-3). Defaults: debug: O=0, release: O=3.
-backend=llvm	Backend combo box	Use LLVM backend (default).
-backend=c	Backend combo box	Use C backend.
-m TRIPLE		Override LLVM target triple to emit to .ll files.
-emit-no-triple		Do not emit any LLVM target triple to .ll files.
-d DATALAYOUT		Override LLVM datalayout to emit to .ll files.
-emit-no-layout		Do not emit any LLVM datalayout to .ll files.
-emit-opt	Build | Options | Emit optimized intermediate code file	Write optimized intermediate code to .opt.ll file.
-quiet	Build | Options | Keep quiet	Output only errors.
-trace		Instrument program/library with tracing enabled.
-debug_heap		Instrument program/library with debug heap enabled.
-no_call_stacks		Do not generate call stack information for exceptions.
-class_dot=FILE	Build | Options | Class hierarchy dot file path.	Generate class hierarchy graph to given dot file FILE.dot (only full config).
-tpg_dot=FILE	Build | Options | Type propagation dot file path.	Generate type propagation graph to given dot file FILE.dot (only full config).
-vcall_dbg		Debug virtual calls (only full config).
-vcall_txt=FILE	Build | Options | Virtual call text file path.	Print devirtualized virtual calls to FILE.txt (only full config).