Cmajor Compiler User's Manual

Seppo Laakko

June 18, 2015

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 Debug Information File
.o	Object Code File
.cml	Cmajor Library File
.cma	Cmajor Assembly File
.dep	Dependency Information File
.exc	Exception Type File
.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. LLVM intermediate code files the project contains are compiled using llc.
7. 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.

8. 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`).

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

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

11. 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.

12. 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.

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`).
-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.
-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.	

4 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]
```

5 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
```