Porting CMake to New Platforms and Languages

In order to generate build files for a particular system, CMake needs to determine what system it is running on, and what compiler tools to use for enabled languages. To do this CMake loads a series of files containing CMake code from the Modules directory. This all has to happen before the first try-compile or try-run is executed. To avoid having to re-compute all of this information for each try-compile and for subsequent runs of CMake, the discovered values are stored in several configured files that are read each time CMake is run. These files are also copied into the try-compile and try-run directories. This chapter will describe how this process of system and tool discovery works. An understanding of the process is necessary to extend CMake to run on new platforms, and to add support for new languages.

11.1 The Determine System Process

The first thing CMake needs to do is to determine what platform it is running on and what the target platform is. Except for when you are cross compiling the host platform and the target platform are identical. The host platform is determined by loading the `CMakeDetermineSystem.cmake` file. On POSIX systems, "uname" is used to get the name of the system. `CMAKE_HOST_SYSTEM_NAME` is set to the result of uname -s, and `CMAKE_HOST_SYSTEM_VERSION` is set to the result of uname -r. On Windows systems, `CMAKE_HOST_SYSTEM_NAME` is set to Windows and `CMAKE_HOST_SYSTEM_VERSION` is set to the value returned by the system function GetVersionEx. The variable `CMAKE_HOST_SYSTEM` is set to a combination of `CMAKE_HOST_SYSTEM_NAME` and `CMAKE_HOST_SYSTEM_VERSION` as follows:

```
${CMAKE_HOST_SYSTEM_NAME}-${CMAKE_HOST_SYSTEM_VERSION}
```

Additionally CMake tries to figure out the processor of the host, on POSIX systems it uses uname -m or uname -p to retrieve this information, on Windows it uses the environment variable `PROCESSOR_ARCHITECTURE`. `CMAKE_HOST_SYSTEM_PROCESSOR` holds the value of the result.

Now that CMake has the information about the host that it is running on, it needs to find this information for the target platform. The results will be stored in the `CMAKE_SYSTEM_NAME`, `CMAKE_SYSTEM_VERSION`, `CMAKE_SYSTEM` and `CMAKE_SYSTEM_PROCESSOR` variables, corresponding to the `CMAKE_HOST_SYSTEM_*` variables described above. See the "Cross compiling with CMake" chapter on how this is done when cross compiling. In all other cases the `CMAKE_SYSTEM_*` variables will be set to the value of their corresponding `CMAKE_HOST_SYSTEM_*` variable.

Once the `CMAKE_SYSTEM` information has been determined, `CMakeSystem.cmake.in` is configured into `${CMAKE_BINARY_DIR}/CMakeFiles/CMakeSystem.cmake`. CMake versions prior to 2.6.0 did not support cross compiling, and so only the `CMAKE_SYSTEM_*` set of variables was available.

11.2 The Enable Language Process

After the platform has been determined, the next step is to enable all languages specified in the `project` command. For each language specified CMake loads `CMakeDetermine(LANG)Compiler.cmake` where `LANG` is the name of the language specified in the `project` command. For example with `project (f Fortran)` the file is called `CMakeDetermineFortranCompiler.cmake`. This file discovers the compiler and tools that will be used to compile files for the particular language. Starting with version 2.6.0 CMake tries to identify the compiler for C, C++ and Fortran not only by its filename, but by compiling some source code, which is named `CMake(LANG)CompilerId.(LANG_SUFFIX)`. If this succeeds, it will return a unique id for every compiler supported by CMake. Once the compiler has been determined for a language, CMake configures the file `CMake(LANG)Compiler.cmake.in` into `CMake(LANG)Compiler.cmake`.

After the platform and compiler tools have been determined, CMake loads `CMakeSystemSpecificInformation.cmake` which in turn will load `${CMAKE_SYSTEM_NAME}.cmake` from the platform subdirectory of modules if it exists for the platform. An example would be `SunOS.cmake`. This file contains OS specific information about compiler flags, creation of executables, libraries, and object files.

Next, CMake loads `CMake(LANG)Information.cmake` for each `LANG` that was enabled. This file in turn loads two files; `${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG-`

`${CMAKE_SYSTEM_PROCESSOR}.cmake` if it exists, and after that `${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG.cmake`. In these file names `COMPILER_ID` references the compiler identification determined as described above. The `CMake(LANG)Information.cmake` file contains default rules for creating executables, libraries, and object files on most UNIX systems. The defaults can be overridden by setting values in either `${CMAKE_SYSTEM_NAME}.cmake` or `${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG.cmake`.

`${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG-${CMAKE_SYSTEM_PROCESSOR}.cmake` is intended to be used only for cross compiling, it is loaded before `${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG.cmake`, so variables can be set up which can then be used in the rule variables.

In addition to the files with the `COMPILER_ID` in their name, CMake also supports these files using the `COMPILER_BASE_NAME`. `COMPILER_BASE_NAME` is the name of the compiler with no path information. For example `cl` would be the `COMPILER_BASE_NAME` for the Microsoft Windows compiler, and `Windows-cl.cmake` would be loaded. If a `COMPILER_ID` exists, it will be preferred over the `COMPILER_BASE_NAME`, since on one side the same compiler can have different names, but there can be also different compilers all with the same name. This means, if

```
${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG-
${CMAKE_SYSTEM_PROCESSOR}.cmake
```

was not found, CMake tries

```
${CMAKE_SYSTEM_NAME}-${COMPILER_BASE_NAME}.cmake
```

and if

```
${CMAKE_SYSTEM_NAME}-${COMPILER_ID}-LANG.cmake
```

was not found, CMake tries

```
${CMAKE_SYSTEM_NAME}-${COMPILER_BASE_NAME}.cmake.
```

`CMake(LANG)Information.cmake` and associated Platform files define special CMake variables, called rule variables. A rule variable consists of a list of commands separated by spaces. The commands are enclosed by quotes. In addition to the normal variable expansion performed by CMake, some special tag variables are expanded by the Makefile generator. Tag

variables have the syntax of <NAME> where NAME is the name of the variable. An example rule variable is CMAKE_CXX_CREATE_SHARED_LIBRARY, and the default setting is

```
set (CMAKE_CXX_CREATE_SHARED_LIBRARY
     "<CMAKE_CXX_COMPILER> <CMAKE_SHARED_LIBRARY_CXX_FLAGS>
      <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS>
      <CMAKE_SHARED_LIBRARY_SONAME_CXX_FLAG><TARGET_SONAME> -o
      <TARGET> <OBJECTS> <LINK_LIBRARIES>")
```

At this point, CMake has determined the system it is running on, the tools it will be using to compile the enabled languages, and the rules to use the tools. This means there is enough information for CMake to perform a try-compile. CMake will now test the detected compilers for each enabled language by loading CMakeTest(LANG)Compiler.cmake. This file will usually run a try-compile on a simple source file for the given language to make sure the chosen compiler actually works.

Once the platform has been determined, and the compilers have been tested, CMake loads a few more files that can be used to change some of the computed values. The first file that is loaded is CMake(PROJECTNAME)Compatibility.cmake where PROJECTNAME is the name given to the top level PROJECT command in the project. The project compatibility file is used to add backwards compatibility fixes into CMake. For example, if a new version of CMake fails to build a project that the previous version of CMake could build, then fixes can be added on a per project basis to CMake. The last file that is loaded is ${CMAKE_USER_MAKE_RULES_OVERRIDE}. This file is an optionally user supplied variable, that can allow a project to make very specific platform based changes to the build rules.

11.3 Porting to a New Platform

Many common platforms are already supported by CMake. However, you may come across a compiler or platform that has not yet been used. If the compiler uses an Integrated Development Environment (IDE), then you will have to extend CMake from the C++ level. However, if the compiler supports a standard make program, then you can specify in CMake the rules to use to compile object code and build libraries by creating CMake configuration files. These files are written using the CMake language with a few special tags that are expanded when the Makefiles are created by CMake. If you run CMake on your system and get a message like the following, you will want to read how to create platform specific settings.

```
System is unknown to cmake, create:
Modules/Platform/MySystem.cmake
to use this system, please send your config file to
cmake@www.cmake.org so it can be added to cmake
```

At a minimum you will need to create the `Platform/${CMAKE_SYSTEM_NAME}.cmake` file for the new platform. Depending on the tools for the platform, you may also want to create `Platform/${CMAKE_SYSTEM_NAME}-${COMPILER_BASE_NAME}.cmake`. On most systems, there is a vendor compiler and the GNU compiler. The rules for both of these compilers can be put in `Platform/${CMAKE_SYSTEM_NAME}.cmake` instead of creating separate files for each of the compilers. For most new systems or compilers, if they follow the basic UNIX compiler flags, you will only need to specify the system specific flags for shared library and module creation.

The following example is from `Platform/IRIX.cmake`. This file specifies several flags, and also one CMake rule variable. The rule variable tells CMake how to use the IRIX CC compiler to create a static library, which is required for template instantiation to work with IRIX CC.

```
# there is no -ldl required on this system
set (CMAKE_DL_LIBS "")

# Specify the flag to create a shared c library
set (CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS
     "-shared -rdata_shared")

# Specify the flag to create a shared c++ library
set (CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS
    "-shared -rdata_shared")

# specify the flag to specify run time paths for shared
# libraries -rpath
set (CMAKE_SHARED_LIBRARY_RUNTIME_C_FLAG "-Wl,-rpath,")

# specify a separator for paths on the -rpath, if empty
# then -rpath will be repeated.
set (CMAKE_SHARED_LIBRARY_RUNTIME_C_FLAG_SEP "")

# if the compiler is not GNU, then specify the initial flags
if (NOT CMAKE_COMPILER_IS_GNUCXX)
  # use the CC compiler to create static library
  set (CMAKE_CXX_CREATE_STATIC_LIBRARY
       "<CMAKE_CXX_COMPILER> -ar -o <TARGET> <OBJECTS>")

  # initializes flags for the native compiler
  set (CMAKE_CXX_FLAGS_INIT "")
  set (CMAKE_CXX_FLAGS_DEBUG_INIT "-g")
  set (CMAKE_CXX_FLAGS_MINSIZEREL_INIT "-O3 -DNDEBUG")
  set (CMAKE_CXX_FLAGS_RELEASE_INIT "-O2 -DNDEBUG")
```

```
  set (CMAKE_CXX_FLAGS_RELWITHDEBINFO_INIT "-O2")
endif (NOT CMAKE_COMPILER_IS_GNUCXX)
```

11.4 Adding a New Language

In addition to porting CMake to new platforms a user may want to add a new language. This can be done either through the use of custom commands, or by defining a new language for CMake. Once a new language is defined, the standard `add_library` and `add_executable` commands can be used to create libraries and executables for the new language. To add a new language, you need to create four files. The name `LANG` has to match in exact case the name used in the `PROJECT` command to enable the language. For example Fortran has the file `CMakeDeterminFortranCompiler.cmake`, and it is enabled with a call like this `project (f Fortran)`. The four files are as follows:

CMakeDetermine(LANG)Compiler.cmake

This file will find the path to the compiler for `LANG` and then configure CMake(LANG)Compiler.cmake.in.

CMake(LANG)Compiler.cmake.in

This file should be used as input to a configure file call in the CMakeDetermine(LANG)Compiler.cmake file. It is used to store compiler information and is copied down into try-compile directories so that try compiles do not need to re-determine and test the `LANG`.

CMakeTest(LANG)Compiler.cmake

This should use a try compile command to make sure the compiler and tools are working. If the tools are working, the following variable should be set in this way:

```
set (CMAKE_(LANG)_COMPILER_WORKS 1 CACHE INTERNAL "")
```

CMake(LANG)Information.cmake

Set values for the following rule variables for `LANG`:

```
CMAKE_(LANG)_CREATE_SHARED_LIBRARY
CMAKE_(LANG)_CREATE_SHARED_MODULE
CMAKE_(LANG)_CREATE_STATIC_LIBRARY
CMAKE_(LANG)_COMPILE_OBJECT
CMAKE_(LANG)_LINK_EXECUTABLE
```

11.5 Rule Variable Listing

For each language that CMake supports, the following rule variables are expanded into build Makefiles at generation time. `LANG` is the name used in the `PROJECT (name LANG)` command. CMake currently supports CXX, C, Fortran, and Java as values for `LANG`.

General Tag Variables

The following set of variables will be expanded by CMake.

<TARGET>

The name of the target being built (this may be a full path).

<TARGET_QUOTED>

The name of the target being built (this may be a full path) double quoted.

<TARGET_BASE>

This is replaced by the name of the target without a suffix.

<TARGET_SONAME>

This is replaced by

CMAKE_SHARED_LIBRARY_SONAME_(LANG)_FLAG

<OBJECTS>

This is the list of object files to be linked into the target.

<OBJECTS_QUOTED>

This is the list of object files to be linked into the target double quoted.

<OBJECT>

This is the name of the object file to be built.

<LINK_LIBRARIES>

This is the list of libraries that are linked into an executable or shared object.

<FLAGS>

This is the command line flags for the linker or compiler.

<LINK_FLAGS>

This is the flags used at link time.

<SOURCE>

The source file name.

Language Specific Information

The following set of variables related to the compiler tools will also be expanded.

<CMAKE_(LANG)_COMPILER>

This is the (LANG) compiler command.

<CMAKE_SHARED_LIBRARY_CREATE_(LANG)_FLAGS>

This is the flags used to create a shared library for (LANG) code.

<CMAKE_SHARED_MODULE_CREATE_(LANG)_FLAGS>

This is the flags used to create a shared module for (LANG) code.

<CMAKE_(LANG)_LINK_FLAGS>

This is the flags used to link a (LANG) program.

<CMAKE_AR>

This is the command to create a .a archive file.

<CMAKE_RANLIB>

This is the command to ranlib a .a archive file.

11.6 Compiler and Platform Examples

Como Compiler

A good example to look at is the como compiler on Linux found in `Modules/Platforms/Linux-como.cmake`. This compiler requires several non-standard commands when creating libraries and executables in order to instantiate C++ templates.

```
# create a shared C++ library

set (CMAKE_CXX_CREATE_SHARED_LIBRARY
    "<CMAKE_CXX_COMPILER> --prelink_objects <OBJECTS>"
    "<CMAKE_CXX_COMPILER>
<CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <LINK_FLAGS> -o <TARGET>
<OBJECTS> <LINK_LIBRARIES>")

# create a C++ static library
```

```
set (CMAKE_CXX_CREATE_STATIC_LIBRARY
    "<CMAKE_CXX_COMPILER> --prelink_objects <OBJECTS>"
    "<CMAKE_AR> cr <TARGET> <LINK_FLAGS> <OBJECTS> "
    "<CMAKE_RANLIB> <TARGET> ")

set (CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_CXX_COMPILER> --prelink_objects <OBJECTS>"
    "<CMAKE_CXX_COMPILER> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>
<FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>")

set (CMAKE_SHARED_LIBRARY_RUNTIME_FLAG "")
set (CMAKE_SHARED_LIBRARY_C_FLAGS "")
set (CMAKE_SHARED_LIBRARY_LINK_FLAGS "")
```

This overrides the creation of libraries (shared and static), and the linking of executable C++ programs. You can see that the linking process of executables and shared libraries requires an extra command that calls the compiler with the flag `--prelink_objects`, and gets all of the object files passed to it.

Borland Compiler

The full Borland compiler rules can be found in `Platforms/Windows-bcc32.cmake`. The following code is an excerpt from that file, showing some of the features used to define rules for the Borland compiler set.

```
set (CMAKE_CXX_CREATE_SHARED_LIBRARY
     "<CMAKE_CXX_COMPILER> ${CMAKE_START_TEMP_FILE}-e<TARGET>
-tWD  <LINK_FLAGS> -tWR <LINK_LIBRARIES>
<OBJECTS>${CMAKE_END_TEMP_FILE}"
 "implib -c -w <TARGET_BASE>.lib <TARGET_BASE>.dll"
)

set (CMAKE_CXX_CREATE_SHARED_MODULE
     ${CMAKE_CXX_CREATE_SHARED_LIBRARY})

# create a C shared library
set (CMAKE_C_CREATE_SHARED_LIBRARY
 "<CMAKE_C_COMPILER> ${CMAKE_START_TEMP_FILE}-e<TARGET> -tWD
<LINK_FLAGS> -tWR <LINK_LIBRARIES>
<OBJECTS>${CMAKE_END_TEMP_FILE}"
 "implib -c -w <TARGET_BASE>.lib <TARGET_BASE>.dll"
)

# create a C++ static library
```

```
set (CMAKE_CXX_CREATE_STATIC_LIBRARY  "tlib
${CMAKE_START_TEMP_FILE}/p512 <LINK_FLAGS> /a <TARGET_QUOTED>
<OBJECTS_QUOTED>${CMAKE_END_TEMP_FILE}")

# compile a C++ file into an object file
set (CMAKE_CXX_COMPILE_OBJECT
    "<CMAKE_CXX_COMPILER>  ${CMAKE_START_TEMP_FILE}-DWIN32 -P
<FLAGS>  -o<OBJECT> -c <SOURCE>${CMAKE_END_TEMP_FILE}")
```

11.7 Extending CMake

Occasionally you will come across a situation where you want to do something during your build process that CMake cannot seem to handle. Examples of this include creating wrappers for C++ classes to make them available to other languages, or creating bindings for C++ classes to support runtime introspection. In these cases you may want to extend CMake by adding your own commands. CMake supports this capability through its C plugin API. Using this API a project can extend CMake to add specialized commands to handle project specific tasks.

A loaded command in CMake is essentially a C code plugin that is compiled into a shared library (a.k.a. DLL). This shared library can then be loaded into the running CMake to provide the functionality of the loaded command. Creating a loaded command is a two step process. You must first write the C code and CMakeLists file for the command, and then place it in your source tree. Next you must modify your project's CMakeLists file to compile the loaded command and load it. We will start by looking at writing the plugin. Before resorting to creating a loaded command you should first see if you can accomplish what you want with a macro. With the commands in CMake a macro/function has almost the same level of flexibility as a loaded command, but does not require compilation or as much complexity. You can almost always, and should, use a macro/function instead of a loaded command.

Creating a Loaded Command

While CMake itself is written in C++ we suggest that you write your plugins using only C code. This avoids a number of portability and compiler issues that can plague C++ plugins being loaded into CMake executables. The API for a plugin is defined in the header file `cmCPluginAPI.h`. This file defines all of the CMake functions that you can invoke from your plugin. It also defines the `cmLoadedCommandInfo` structure that is passed to a plugin. Before going into detail about these functions, consider the following simple plugin:

```
#include "cmCPluginAPI.h"

static int InitialPass(void *inf, void *mf,
                       int argc, char *argv[])
```

```
{
  cmLoadedCommandInfo *info = (cmLoadedCommandInfo *)inf;
  info->CAPI->AddDefinition(mf, "FOO", "BAR");

  return 1;
}
void CM_PLUGIN_EXPORT
HELLO_WORLDInit(cmLoadedCommandInfo *info)
{
  info->InitialPass = InitialPass;
  info->Name = "HELLO_WORLD";
}
```

First this plugin includes the `cmCPluginAPI.h` file to get the definitions and structures required for a plugin. Next it defines a static function called `InitialPass` that will be called whenever this loaded command is invoked. This function is always passed four parameters: the `cmLoadedCommandInfo` structure, the Makefile, the number of arguments, and the list of arguments. Inside this function we typecast the `inf` argument to its actual type and then use it to invoke the C API (CAPI) `AddDefinition` function. This function will set the variable `FOO` to the value of `BAR` in the current `cmMakefile` instance.

The second function is called `HELLO_WORLDInit` and it will be called when the plugin is loaded. The name of this function must exactly match the name of the loaded command with `Init` appended. In this example the name of the command is `HELLO_WORLD` so the function is named `HELLO_WORLDInit`. This function will be called as soon as your command is loaded. It is responsible for initializing the elements of the `cmLoadedCommandInfo` structure. In this example it sets the `InitialPass` member to the address of the `InitialPass` function defined above. It will then set the name of the command by setting the `Name` member to `"HELLO_WORLD"`.

Using a Loaded Command

Now let us consider how to use this new `HELLO_WORLD` command in a project. The basic process is that CMake will have to compile the plugin into a shared library and then dynamically load it. To do this you first create a subdirectory in your project's source tree called CMake or CMakeCommands (by convention, any name can be used). Place the source code to your plugin in that directory. We recommend naming the file with the prefix `cm` and then the name of the command. For example, `cmHELLO_WORLD.c`. Then you must create a simple CMakeLists.txt file for this directory that includes instructions to build the shared library. Typically this will be the following:

```
project (HELLO_WORLD)

set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}"
```

```
  "${CMAKE_ANSI_CXXFLAGS}"
  )

set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS}"
  "${CMAKE_ANSI_CFLAGS}"
  )
include_directories (${CMAKE_ROOT}/include
  ${CMAKE_ROOT}/Source
  )

add_library (cmHELLO_WORLD MODULE cmHELLO_WORLD.c)
```

It is critical that you name the library `cm` followed by the name of the command as shown in the `add_library` call in the above example (e.g. `cmHELLO_WORLD`). When CMake loads a command it assumes that the command is in a library named using that pattern. The next step is to modify your project's main CMakeLists file to compile and load the plugin. This can be accomplished with the following code:

```
# if the command has not been loaded, compile and load it
if (NOT COMMAND HELLO_WORLD)

  # try compiling it first
  try_compile (COMPILE_OK
    ${PROJECT_BINARY_DIR}/CMake
    ${PROJECT_SOURCE_DIR}/CMake
    HELLO_WORLD
    )

  # if it compiled OK then load it
  if (COMPILE_OK)
    load_command (HELLO_WORLD
      ${PROJECT_BINARY_DIR}/CMake
      ${PROJECT_BINARY_DIR}/CMake/Debug
      )

  # if it did not compile OK, then display an error
  else (COMPILE_OK)
     message ("error compiling HELLO_WORLD extension")
  endif (COMPILE_OK)

endif (NOT COMMAND HELLO_WORLD)
```

In the above example you would simply replace `HELLO_WORLD` with the name of your command and replace `${PROJECT_SOURCE_DIR}/CMake` with the actual name of the subdirectory where you placed your loaded command. Now let us look at creating loaded commands in more detail. We will start by looking at the `cmLoadedCommandInfo` structure.

```
typedef const char* (*CM_DOC_FUNCTION)();

typedef int (*CM_INITIAL_PASS_FUNCTION)(
    void *info, void *mf, int argc, char *[]);

typedef void (*CM_FINAL_PASS_FUNCTION)(
    void *info, void *mf);
typedef void (*CM_DESTRUCTOR_FUNCTION)(void *info);

typedef struct {
    unsigned long reserved1;
    unsigned long reserved2;
    cmCAPI *CAPI;
    int m_Inherited;
    CM_INITIAL_PASS_FUNCTION InitialPass;
    CM_FINAL_PASS_FUNCTION FinalPass;
    CM_DESTRUCTOR_FUNCTION Destructor;
    CM_DOC_FUNCTION GetTerseDocumentation;
    CM_DOC_FUNCTION GetFullDocumentation;
    const char *Name;
    char *Error;
    void *ClientData;
  } cmLoadedCommandInfo;
```

The first two entries of the structure are reserved for future use. The next entry, CAPI, is a pointer to a structure containing pointers to all the CMake functions you can invoke from a plugin. The `m_Inherited` member only applies to CMake versions 2.0 and earlier. It can be set to indicate if this command should be inherited by subdirectories or not. If you are creating a command that will work with versions of CMake prior to 2.2 then you probably want to set this to zero. The next five members are pointers to functions that your plugin may provide. The `InitialPass` function must be provided and it is invoked whenever your loaded command is invoked from a CMakeLists file. The `FinalPass` function is optional and is invoked after configuration but before generation of the output. The `Destructor` function is optional and will be invoked when your command is destroyed by CMake (typically on exit). It can be used to clean up any memory that you have allocated in the `InitialPass` or `FinalPass`. The next two functions are optional and are used to provide documentation for your command. The `Name` member is used to store the name of your command. This is what will be compared against when parsing a CMakeLists file, it should be in all caps in keeping

with CMake's naming conventions. The `Error` and `ClientData` members are used internally by CMake, you should not directly access them. Instead you can use the CAPI functions to manipulate them.

Now let us consider some of the common CAPI functions you will use from within a loaded command. First we will consider some utility functions that are provided specifically for loaded commands. Since loaded commands use a C interface they will receive arguments as `(int argc, char *argv[])`, for convenience you can call `GetTotalArgumentSize(argc, argv)` which will return the total length of all the arguments. Likewise some CAPI methods will return an `(argc,argv)` pair that you will be responsible for freeing. The `FreeArguments(argc, argv)` function can be used to free such return values. If your loaded command has a `FinalPass()` then you might want to pass data from the `InitialPass()` to the `FinalPass()` invocation. This can be accomplished using the `SetClientData(void *info, void *data)` and `void *GetClientData(void *info)` functions. Since the client data is passed as a void * argument, any client data larger than a pointer must be allocated and then finally freed in your `Destructor()` function. Be aware that CMake will create multiple instances of your loaded command so using global variables or static variables is not recommended. If you should encounter an error in executing your loaded command, you can call `SetError(void *info, const char *errorString)` to pass an error message on to the user.

Another group of CAPI functions worth noting are the `cmSourceFile` functions. `cmSourceFile` is a C++ object that represents information about a single file including its full path, file extension, special compiler flags, etc. Some loaded commands will need to either create or access `cmSourceFile` instances. This can be done using the `void *CreateSourceFile()` and `void * GetSource (void *mf, const char *sourceName)` functions. Both of these functions return a pointer to a `cmSourceFile` as a `void *` return value. This pointer can then be passed into other functions that manipulate `cmSourceFiles` such as `SourceFileGetProperty()` or `SourceFileSetProperty()`.