Update for LLVM Version 3.5

In this chapter, we will cover the changes introduced with the LLVM 3.5 release of September 4, 2014. This represents almost 9 months of development since the last 3.4 release.

The most important changes from LLVM 3.4 to LLVM 3.5, regarding the code changes in our examples, are the following:

- The removal of the `OwningPtr<>` class template in favor of the standard `std::unique_ptr<>` class template of C++ libraries (C++11). Since LLVM requires a C++11-compliant compiler, which supports this template, there is no point maintaining a separate LLVM template to replicate the work of the C++ standard library. To update your code, you must remove includes that refer to the `llvm/ADT/OwningPtr.h` header and substitute your `OwningPtr<>` instances for the `std::unique_ptr<>` ones.
- Interface updates to return a `std::unique_ptr<>` instance in lieu of a naked pointer. This is part of an effort to produce safer C++ code and is still work in progress. You should see more interfaces being converted to use `std::unique_ptr<>` objects in the future LLVM versions.
- Interface updates to use a more modern error-reporting mechanism. Previously, many LLVM functions relied on a pointer to a string object and, if the function failed to do its work, it would write the error message in this string object. LLVM programmers are making an effort to convert these functions to abandon the error string and return a `std::error_code` object instead. Some functions might also use `llvm::ErrorOr<T>` as a return type, which is a convenient way to return either a `T` object or a nonzero `std::error_code` object, in case of an error. You should see more interface changes in this direction in future LLVM versions.

In the following sections of this chapter, we will revisit chapters from the book and briefly present the aspects that changed with LLVM 3.5, starting with the first chapter.

Updates for chapter 1

In *Chapter 1, Build and Install LLVM*, of this book deals with LLVM installation and, while the concepts of this chapter are still valid, many download links have changed. In this section, we will provide a thorough review of these new links to LLVM 3.5, and will also update command-line snippets to match LLVM 3.5.

Obtaining the official prebuilt binaries

We have an updated list of prebuilt packages for Version 3.5, which can be downloaded from the official LLVM website. To view all the options for different versions of LLVM, go to `http://www.llvm.org/releases/download.html` and see the **Pre-built Binaries** section relative to the version you want to download.

Architecture	**Version**
x86_64	Ubuntu 14.04, Fedora 20, FreeBSD 10, Mac OS X 10.9, Windows, and openSUSE 13.1
i386	FreeBSD 9.2, Fedora 20, and openSUSE 13.1
ARMv7a	Linux-generic
AArch64	GNU Linux
MIPS	GNU Linux

Staying updated with the snapshot packages

Notice that the snapshot packages are synced with the LLVM subversion repository. After an LLVM version is released, the version of the trunk in the repository is immediately incremented to the next one. For example, if you work with snapshot packages or with the SVN trunk version, you will notice that Clang and LLVM now report themselves as *Version 3.6*, even though this version is still in its early development stage. This is used to make you remember that the trunk version might have features that, in a stable version, will only be available in the next release.

Debian/Ubuntu Linux

We have a small update on the following sequence of commands to reflect the aforementioned version change:

```
$ sudo echo "deb http://llvm.org/apt/trusty/ llvm-toolchain-trusty main"
>> /etc/apt/sources.list
$ wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add
-
```

```
$ sudo apt-get update
$ sudo apt-get install clang-3.6 llvm-3.6
```

Notice that in comparison with the sequence of commands from *Chapter 1, Build and Install LLVM*, we only changed `clang-3.5` and `llvm-3.5` to `clang-3.6` and `llvm-3.6`, respectively. We also updated the name of the Ubuntu distribution to `trusty`. However, you should use the appropriate choice for you. Refer to `http://llvm.org/apt` for more choices of Debian and Ubuntu versions.

Obtaining sources

To download the sources from the 3.5 release, you can either go to the website, `http://llvm.org/releases/download.html#3.5`, or directly download and prepare the sources for compilation as shown in the next command lines. In the following instructions, we only updated the links, although the change is more substantial than simple number-related changes. LLVM packaging is not consistent across versions, and not only the name of the package but also the compression format is different for different versions. Version 3.5 is referred in packages as 3.5.0 because the community now expects intermediary point releases, and the last zero marks the first version prior to any point release. The compression format for the files is now `.tar.xv` (LZMA). Use the following commands to download and install LLVM, Clang, and Clang extra tools:

```
$ wget http://llvm.org/releases/3.5.0/llvm-3.5.0.src.tar.xz
$ wget http://llvm.org/releases/3.5.0/cfe-3.5.0.src.tar.xz
$ wget http://llvm.org/releases/3.5.0/clang-tools-extra-3.5.0.src.tar.xz
$ tar xf llvm-3.5.0.src.tar.xz; tar xf cfe-3.5.0.src.tar.xz
$ tar xf clang-tools-extra-3.5.0.src.tar.xz
$ mv llvm-3.5.0.src llvm
$ mv cfe-3.5.0.src llvm/tools/clang
$ mv clang-tools-extra-3.5.0.src llvm/tools/clang/tools/extra
```

SVN

If you want to use the 3.5 stable version, substitute `trunk` for `tags/RELEASE_350/final` in all the commands presented in the SVN section of *Chapter 1, Build and Install LLVM*.

Updates for chapter 2

Similar to the previous section, most changes in *Chapter 2*, *External Projects*, are related to updated links. In this section, we will present all new links and command snippets to install external LLVM projects. We finish this section with an example that teaches you how to build and install a complete Clang/LLVM 3.5 package, putting together the topics of the first two chapters of the book.

Building and installing Clang extra tools

You can obtain an official snapshot of Version 3.5 of this project at `http://llvm.org/releases/3.5.0/clang-tools-extra-3.5.0.src.tar.gz`. To compile this set of tools without difficulty, build it together with the source of core LLVM and Clang, relying on the LLVM build system. To do this, put the source directory into the Clang source tree as follows:

```
$ wget http://llvm.org/releases/3.5.0/clang-tools-extra-3.5.0.src.tar.xz
$ tar xf clang-tools-extra-3.5.0.src.tar.xz
$ mv clang-tools-extra-3.5.0.src llvm/tools/clang/tools/extra
```

Understanding Compiler-RT

You can download Compiler-RT Version 3.5 at `http://llvm.org/releases/3.5.0/compiler-rt-3.5.0.src.tar.xz` or look for more versions at `http://llvm.org/releases/download.html`.

We also have an updated command sequence for this section:

```
$ wget http://llvm.org/releases/3.5.0/compiler-rt-3.5.0.src.tar.xz
$ tar xf compiler-rt-3.5.0.src.tar.xz
$ mv compiler-rt-3.5.0.src llvm/projects/compiler-rt
```

Understanding the LLVM test suite

You can find the sources for LLVM Version 3.5 test suite at `http://llvm.org/releases/3.5.0/test-suite-3.5.0.src.tar.xz`.

To fetch the sources, use the following commands:

```
$ wget http://llvm.org/releases/3.5.0/test-suite-3.5.0.src.tar.xz
$ tar xf test-suite-3.5.0.src.tar.xz
$ mv test-suite-3.5.0.src llvm/projects/test-suite
```

Using LLDB

You can obtain the sources for LLDB 3.5 at `http://llvm.org/releases/3.5.0/lldb-3.5.0.src.tar.xz`. We also have an updated sequence of commands for LLDB 3.5:

```
$ wget http://llvm.org/releases/3.5.0/lldb-3.5.0.src.tar.xz
$ tar xf lldb-3.5.0.src.tar.xz
$ mv lldb-3.5.0.src llvm/tools/lldb
```

Note that besides being built with a reusable C++ API, you can also install LLDB Python bindings. With this feature, you can write your own Python scripts that rely on LLDB to leverage your debugging experience. To build LLDB with Python bindings, make sure you have the `python-dev`, `libedit-dev`, and `swig` packages installed.

Introducing the libc++ standard library

To build libc++ 3.5 with libsupc++ in a GNU/Linux machine, download the source packages with the following commands:

```
$ wget http://llvm.org/releases/3.5.0/libcxx-3.5.0.src.tar.xz
$ tar xf libcxx-3.5.0.src.tar.xz
$ mv libcxx-3.5.0.src libcxx
```

In Version 3.5, you can put the `libcxx` and `libcxxabi` folders in the `projects` subfolder of the LLVM source code and configure LLVM with CMake. If you do this, when you install LLVM, it will also install libc++ and libc++abi.

Summarizing the updates for chapters 1 and 2

To summarize the updates for chapters 1 and 2, we will now present a complete example that builds and installs LLVM, Clang, Clang extra tools, Compiler-RT, and libc++ 3.5 from scratch by using `git` to fetch the sources—yet another way to obtain them.

Exercising a complete Clang 3.5 build and install example

In this example, we make use of the `cmake` command. Refer to *Chapter 1*, *Build and Install LLVM*, for a guide with instructions on how to pick your own CMake parameters. It is also important to point out that you might have problems if you use the newer CMake 3.0 in this build, and hence, we rely on CMake 2.8.12 (available in Ubuntu Trusty). Even though this example was tested in Ubuntu Trusty (14.04), the only Ubuntu-dependent line is the first one, which will install the prerequisites if you do not have them already.

If you are using an older system whose default compiler is outdated, remember to either update your system or install a recent compiler and configure it in the `CC` and `CXX` environment variables before attempting to build LLVM. For instance, when building on Ubuntu 12.04, you need to install g++-4.8 and set the `CXX` environment variable to `g++-4.8` before running CMake. When building LLVM, it is always advisable to make sure your system has at least 2 GB of RAM because the system might run out of memory when linking the extensive set of LLVM libraries.

The build example is as follows:

```
$ sudo apt-get install build-essential zlib1g-dev python ninja-build
cmake git
$ git clone http://llvm.org/git/llvm.git -b release_35
$ git clone http://llvm.org/git/clang.git -b release_35 llvm/tools/clang
$ git clone http://llvm.org/git/clang-tools-extra.git -b release_35 llvm/
tools/clang/tools/extra
$ git clone http://llvm.org/git/compiler-rt.git -b release_35 llvm/
projects/compiler-rt
$ git clone http://llvm.org/git/libcxx.git -b release_35 llvm/projects/
libcxx
$ mkdir -p build && cd build
$ cmake ../llvm -GNinja -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_
PREFIX=$(pwd)/../install -DLLVM_TARGETS_TO_BUILD=X86
$ ninja && ninja install
$ cd ..
```

You can now test all this by using the Clang C++ compiler to compile a simple program:

```
$ install/bin/clang++ -o test -x c++ - <<< '#include <iostream>
> int main() { std::cout << "Hello World!\n"; return 0; }'
$ ./test
```

We used the dash flag (`-`) to indicate that we will supply the entire program via the standard input, and then we used a character sequence to indicate that the contents of the standard input for this command follows the `<<<` string. Since the program is supplied via the standard input, Clang cannot rely on the file extension to determine the file language, and to solve this, we used the `-x c++` flag that sets the input to C++. Notice that you must hit the *Enter* key after `#include <iostream>`, and then complete the command in the next line. This is a good way to make quick tests with Clang without saving the input into a file.

Clang will link this program with your regular C++ library. If you want libc++, you should use quite a few extra flags. Suppose that our hello-world example is saved to `hello.cpp` (you can use the standard input here as well, if you want to):

```
$ install/bin/clang++ hello.cpp -o test -stdlib=libc++ -nodefaultlibs
-lc++ -lsupc++ -lm -lc -lgcc_s -lgcc
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$(pwd)/install/lib
$ ./test
```

If you choose to also build `libc++abi` to substitute `libsupc++`, use `-lc++abi` instead of `-lsupc++` in the preceding code. However, we do not put `libc++abi` in this example because Version 3.5 does not compile cleanly in GCC 4.8.2 for Linux. If you want to build it, just put its folder in the `projects` subfolder alongside the `libcxx` folder.

Now, you should have a complete, modern Clang-based C++ compiler at your disposal. If you want to test your Clang with the newest C++ standard, remember to add the `-std=c++14` flag when compiling your C++ sources.

Running all examples of this book with a Docker image

We prepared a Docker image with Clang and LLVM installed in the same way as we saw in the preceding sections. Docker is a container manager platform that allows us to ship a container with all the necessary prerequisites to run LLVM, Clang, and our code examples. Thus, our image also contains all code examples from this book. To use this image, you first need to install Docker. Check `https://docs.docker.com/installation` for details. The installation instructions for Ubuntu, for example, are the following:

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
$ sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/
docker.io
```

After Docker is installed, run our image `rafaelauler/llvmbook` as follows:

```
$ sudo docker run -t -i rafaelauler/llvmbook /bin/bash
```

Docker will pull the image, which is about 1.2 GB in size, from Docker Hub. You can then run bash in this container that already has everything you need to test the examples of this book. You can find the examples and LLVM itself in the `/workspace` folder. Before running Clang or LLVM tools, you need to set the `PATH` and `LD_LIBRARY_PATH` environment variables to the correct locations. To show you how this works, we will present how to compile the first example from *Chapter 3*, *Tools and Design*:

```
$ sudo docker run -t –i rafaelauler/llvmbook /bin/bash
root@44ff5a0b9ad9:/# export PATH=$PATH:/workspace/install/bin
root@44ff5a0b9ad9:/# export LD_LIBRARY_PATH=/workspace/install/lib
root@44ff5a0b9ad9:/# cd /workspace/examples/Ch3
root@44ff5a0b9ad9:/workspace/examples/Ch3# make
root@44ff5a0b9ad9:/workspace/examples/Ch3# ./helloworld -help
```

Updates for chapter 3

In this and the following sections, we will mostly concentrate on code changes, but the concepts remain the same. In *Chapter 3*, *Tools and Design*, we have an updated first project for LLVM but, most importantly, we have an updated Makefile that is used throughout many examples in the book. We will first present the updated Makefile followed by the code.

Writing your first LLVM project

We present the updated Makefile in this section, highlighting the changes that we made. If you copy the code from the book instead of downloading from the publisher's website, remember to use the tab character before the rule commands, as GNU Make makes distinctions between tabs and spaces.

We update the Makefile to include an extra `-fno-rtti` flag to guarantee that you will not suffer with link errors, since the LLVM library is compiled without RTTI, but `llvm-config --cxxflags` is not enough to provide this flag. We also add two new variables, `LLVMLIBS` and `SYSTEMLIBS`, to improve the linker command-line organization. While `LLVMLIBS` is added for cosmetic purposes, `SYSTEMLIBS` is critical, and we use it to specify system libraries such as `pthreads` that LLVM rely on and need to link correctly.

The contents of the updated Makefile are the following:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
LLVMLIBS=$(shell $(LLVM_CONFIG) --libs bitreader support)
SYSTEMLIBS=$(shell $(LLVM_CONFIG) --system-libs)

HELLO=helloworld
HELLO_OBJECTS=hello.o

default: $(HELLO)

%.o : $(SRC_DIR)/%.cpp
  @echo Compiling $*.cpp
  $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(HELLO) : $(HELLO_OBJECTS)
  @echo Linking $@
  $(QUIET)$(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ $(LLVMLIBS)
$(SYSTEMLIBS)

clean::
  $(QUIET)rm -f $(HELLO) $(HELLO_OBJECTS)
```

Next, we present the updated code for the example of this chapter, with changes highlighted. We remove include directives for `Support/raw_os_ostream.h`, `Support/system_error.h`, and `<iostream>`. The `system_error.h` header is removed from our include directives to reflect an updated error-handling interface, which now relies on the new `llvm::ErrorOr<>` class template. This class template creates a type that represents an `std::error_code` object if the function has an error, and an object specified by the template if the function is successfully executed. We use the `getError()` member function to check if the value of `std::error_code` is zero and, if positive, there is no error and we can retrieve the object via `operator*()` or the `get()` member function.

In this example update, we also take the opportunity to present you a different way to write the same code. We remove streams with header files to show you how to write to the standard output with streams, without including `<iostream>`. To do this, we simply use the `llvm::outs()` object. The contents of the updated code are the following:

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorOr.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Bitcode file"),
         cl::Required);

int main(int argc, char** argv)
{
  cl::ParseCommandLineOptions(argc, argv, "LLVM hello world\n");
  LLVMContext context;

  ErrorOr<std::unique_ptr<MemoryBuffer>> mb =
    MemoryBuffer::getFile(FileName);
  if (std::error_code ec = mb.getError()) {
    errs() << ec.message();
    return -1;
  }

  ErrorOr<Module *> m = parseBitcodeFile(mb->get(), context);
  if (std::error_code ec = m.getError()) {
    errs() << "Error reading bitcode: " << ec.message() << "\n";
    return -1;
  }

  for (Module::const_iterator I = (*m)->getFunctionList().begin(),
    E = (*m)->getFunctionList().end(); I != E; ++I) {
    if (!I->isDeclaration()) {
      outs() << I->getName() << " has " << I->size()
             << " basic blocks.\n";
```

```
        }
      }

      return 0;
    }
```

Notice that you might produce an LLVM bitcode to test this program using our own code:

```
$ clang++ -c $(llvm-config --cppflags) $(llvm-config --cxxflags) -emit-llvm hello.cpp -o mysource.bc
$ ./helloworld mysource.bc
```

Updates for chapter 4

In *Chapter 4*, *The Frontend*, our first three examples do not need to change. The reason is because we relied on the Clang C API, which is stable and should not change across LLVM releases. Therefore, the good thing is that the code is still perfectly compatible.

We reproduce the updated Makefile for Clang-based projects (*Chapter 4*, *The Frontend* and *Chapter 10*, *Clang Tools with LibTooling*) in the following code, with changes highlighted. We include the same `-fno-rtti` and `SYSTEMLIBS` tags from the previous section. The contents of the updated Makefile are the following:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
CLANGLIBS=\
  -Wl,--start-group\
  -lclang\
  -lclangFrontend\
  -lclangDriver\
  -lclangSerialization\
  -lclangParse\
  -lclangSema\
    -lclangAnalysis\
    -lclangEdit\
    -lclangAST\
    -lclangLex\
    -lclangBasic\
    -Wl,--end-group
LLVMLIBS=$(shell $(LLVM_CONFIG) --libs)
SYSTEMLIBS=$(shell $(LLVM_CONFIG) --system-libs)

PROJECT=myproject
PROJECT_OBJECTS=project.o

default: $(PROJECT)

%.o : $(SRC_DIR)/%.cpp
  @echo Compiling $*.cpp
  $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(PROJECT) : $(PROJECT_OBJECTS)
  @echo Linking $@
  $(QUIET)$(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ $(CLANGLIBS)
$(LLVMLIBS) $(SYSTEMLIBS)

clean::
  $(QUIET)rm -f $(PROJECT) $(PROJECT_OBJECTS)
```

Next, we show the updated code for our final example in this chapter, the simple driver, with changes highlighted. We replace an `IntrusiveRefCntPtr<>` instance with the `std::shared_ptr<>` instance, another template to count references in a smart pointer way. The reason to do this is because `IntrusiveRefCntPtr<T>`, as the name implies, requires the `T` class to provide specific member functions to aid in the task of managing references, and `TargetOptions` no longer supports this. Since we will not use the `IntrusiveRefCntPtr<>` class template, we remove `ADT/IntrusiveRefCntPtr.h` from our include list.

We also make minor interface changes; for example, `CompilerInstance::createPreprocessor()` now requires a `TranlationUnitKind` object, and the convenience function `clang::SourceManager::createMainFileID()` is removed, requiring us to separately call `createFileID()`, and then `setMainFileID()`. The contents of the updated code for `project.cpp` are the following:

```
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Host.h"
#include "clang/AST/ASTContext.h"
```

```
#include "clang/AST/ASTConsumer.h"
#include "clang/Basic/Diagnostic.h"
#include "clang/Basic/DiagnosticOptions.h"
#include "clang/Basic/FileManager.h"
#include "clang/Basic/SourceManager.h"
#include "clang/Basic/LangOptions.h"
#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/TargetOptions.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/TextDiagnosticPrinter.h"
#include "clang/Lex/Preprocessor.h"
#include "clang/Parse/Parser.h"
#include "clang/Parse/ParseAST.h"
#include <iostream>

using namespace llvm;
using namespace clang;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"), cl::Required);

int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv, "My simple driver\n");
  CompilerInstance CI;
  DiagnosticOptions diagnosticOptions;
  CI.createDiagnostics();

  std::shared_ptr<TargetOptions> PTO(new TargetOptions());
  PTO->Triple = sys::getDefaultTargetTriple();
  TargetInfo *PTI =
    TargetInfo::CreateTargetInfo(CI.getDiagnostics(), PTO);
  CI.setTarget(PTI);

  CI.createFileManager();
  CI.createSourceManager(CI.getFileManager());
  CI.createPreprocessor(TU_Complete);
  CI.getPreprocessorOpts().UsePredefines = false;
  ASTConsumer *astConsumer = CreateASTPrinter(NULL, "");
  CI.setASTConsumer(astConsumer);

  CI.createASTContext();
  CI.createSema(TU_Complete, NULL);
```

```
    const FileEntry *pFile = CI.getFileManager().getFile(FileName);
    if (!pFile) {
      std::cerr << "File not found: " << FileName << std::endl;
      return 1;
    }
    CI.getSourceManager().setMainFileID(
        CI.getSourceManager().createFileID(
        pFile, SourceLocation(), SrcMgr::C_User));
    CI.getDiagnosticClient().BeginSourceFile(CI.getLangOpts(), 0);
    ParseAST(CI.getSema());
    CI.getASTContext().PrintStats();
    CI.getDiagnosticClient().EndSourceFile();
    return 0;
  }
```

Updates for chapter 5

In *Chapter 5, The LLVM Intermediate Representation*, of the book, we discuss the code to generate LLVM IR on the fly. This code has only minor modifications to link with the newer LLVM 3.5. We replace an `OwningPtr<>` instance with the `std::unique_ptr<>` instance, and we also remove the verifier and the `PrintModulePass` headers to make this example simpler. However, if you want to keep them in the newer LLVM 3.5, they will have to be moved from `llvm/Analysis/Verifier.h` to `llvm/IR/Verifier.h` and from `llvm/Assembly/PrintModulePass.h` to `llvm/IR/PrintModulePass.h`, respectively.

We also add an include directive for `Support/FileSystem.h` to supply a missing declaration after the removal of the aforementioned headers. The full code to generate LLVM IR on the fly is as follows, with changes highlighted:

```
#include <llvm/ADT/SmallVector.h>
#include <llvm/IR/BasicBlock.h>
#include <llvm/IR/CallingConv.h>
#include <llvm/IR/Function.h>
#include <llvm/IR/Instructions.h>
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/Bitcode/ReaderWriter.h>
#include <llvm/Support/FileSystem.h>
#include <llvm/Support/ToolOutputFile.h>

using namespace llvm;
```

```
Module *makeLLVMModule() {
  Module *mod = new Module("sum.ll", getGlobalContext());
  SmallVector<Type*, 2> FuncTyArgs;
  FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
  FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
  FunctionType *FuncTy = FunctionType::get(IntegerType::get
                                         (mod->getContext(), 32),
                                          FuncTyArgs, false);
  Function *funcSum =
    Function::Create(FuncTy, GlobalValue::ExternalLinkage, "sum",
mod);
  funcSum->setCallingConv(CallingConv::C);
  Function::arg_iterator args = funcSum->arg_begin();
  Value *int32_a = args++;
  int32_a->setName("a");
  Value *int32_b = args++;
  int32_b->setName("b");

  BasicBlock *labelEntry =
    BasicBlock::Create(mod->getContext(), "entry", funcSum, 0);
  AllocaInst *ptrA =
    new AllocaInst(IntegerType::get(mod->getContext(), 32), "a.addr",
                   labelEntry);
  ptrA->setAlignment(4);
  AllocaInst *ptrB =
    new AllocaInst(IntegerType::get(mod->getContext(), 32), "b.addr",
                   labelEntry);
  ptrB->setAlignment(4);

  StoreInst *st0 = new StoreInst(int32_a, ptrA, false, labelEntry);
  st0->setAlignment(4);
  StoreInst *st1 = new StoreInst(int32_b, ptrB, false, labelEntry);
  st1->setAlignment(4);

  LoadInst *ld0 = new LoadInst(ptrA, "", false, labelEntry);
  ld0->setAlignment(4);
  LoadInst *ld1 = new LoadInst(ptrB, "", false, labelEntry);
  ld1->setAlignment(4);

  BinaryOperator *addRes = BinaryOperator::Create(Instruction::Add,
ld0, ld1,
                                                 "add", labelEntry);
  ReturnInst::Create(mod->getContext(), addRes, labelEntry);
```

```
    return mod;
  }

  int main(int argc, char **argv) {
    Module *Mod = makeLLVMModule();
    std::string ErrorInfo;
    std::unique_ptr<tool_output_file> Out(new tool_output_file(
        "./sum.bc", ErrorInfo, sys::fs::F_None));
    if (!ErrorInfo.empty()) {
      errs() << ErrorInfo << "\n";
      return -1;
    }
    WriteBitcodeToFile(Mod, Out->os());
    Out->keep();
    return 0;
  }
```

In this chapter, we also present a Makefile to build a dynamically loadable pass. We make a minor update to this Makefile to use the `-fno-rtti` flag. This change is highlighted in the following code:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)

ifeq ($(shell uname),Darwin)
LOADABLE_MODULE_OPTIONS=-bundle -undefined dynamic_lookup
else
LOADABLE_MODULE_OPTIONS=-shared -Wl,-O1
endif

FNARGPASS=fnarg.so
FNARGPASS_OBJECTS=FnArgCnt.o

default: $(FNARGPASS)
```

```
%.o : $(SRC_DIR)/%.cpp
  @echo Compiling $*.cpp
  $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(FNARGPASS) : $(FNARGPASS_OBJECTS)
  @echo Linking $@
  $(QUIET)$(CXX) -o $@ $(LOADABLE_MODULE_OPTIONS) $(CXXFLAGS)
$(LDFLAGS) $^

clean::
  $(QUIET)rm -rf $(FNARGPASS_OBJECTS) $(FNARGPASS)
```

Updates for chapter 7

The updates for the code of *Chapter 7*, *The Just-in-Time Compiler*, regarding the JIT infrastructure, only reflect in the removal of the `OwningPtr<>` class template and newer error-handling interfaces. Keep in mind that LLVM 3.5 will be the last release to support the older JIT infrastructure.

The updated code for our first example is reproduced next with changes highlighted. Notice that we remove the declaration of `std::string ErrorMessage`, which is unnecessary in this more modern error-handling interface. We also include the `ErrorOr.h` header while removing the `OwningPtr.h` and `system_error.h` headers. Overall, we update the code in a similar way to the update seen in the code example from *Chapter 3*, *Tools and Design*. The contents of the code are the following:

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/ExecutionEngine/JIT.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/ErrorOr.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/TargetSelect.h"

using namespace llvm;

int main() {
  InitializeNativeTarget();
  LLVMContext Context;
  ErrorOr<std::unique_ptr<MemoryBuffer>> Buffer =
    MemoryBuffer::getFile("./sum.bc");
  if (Buffer.getError()) {
    errs() << "sum.bc not found\n";
    return -1;
  }

  ErrorOr<Module *> M = parseBitcodeFile(Buffer->get(), Context);
  if (std::error_code ec = M.getError()) {
    errs() << "Error reading bitcode: " << ec.message() << "\n";
    return -1;
  }

  std::unique_ptr<ExecutionEngine> EE(EngineBuilder(*M).create());

  Function *SumFn = (*M)->getFunction("sum");
  int (*Sum)(int, int) =
    (int (*)(int,int)) EE->getPointerToFunction(SumFn);
  int res = Sum(4,5);
  outs() << "Sum result: " << res << "\n";
  res = Sum(res, 6);
  outs() << "Sum result: " << res << "\n";

  EE->freeMachineCodeForFunction(SumFn);
  llvm_shutdown();
  return 0;
}
```

To compile this, you can copy the updated Makefile from *Chapter 3, Tools and Design*, and use `LLVMLIBS`:

```
LLVMLIBS=$(shell $(LLVM_CONFIG) --libs jit mcjit native irreader)
```

You will also need to update the variables and filenames. However, to compile from the command line, run:

```
clang++ sum-jit.cpp -g -rdynamic -fno-rtti $(llvm-config --cppflags
--cxxflags --ldflags --libs jit mcjit native irreader --system-libs) -o
sum-jit
```

Next, we present the updated code that uses `GenericValue`. The changes to comply with LLVM 3.5 are the same as that of the previous code. Check them in the following code:

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/ExecutionEngine/GenericValue.h"
#include "llvm/ExecutionEngine/JIT.h"
```

```
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/ErrorOr.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/TargetSelect.h"

using namespace llvm;

int main() {
  InitializeNativeTarget();
  LLVMContext Context;

  ErrorOr<std::unique_ptr<MemoryBuffer>> Buffer =
MemoryBuffer::getFile("./sum.bc");
  if (Buffer.getError()) {
    errs() << "sum.bc not found\n";
    return -1;
  }

  ErrorOr<Module *> M = parseBitcodeFile(Buffer->get(), Context);
  if (std::error_code ec = M.getError()) {
    errs() << "Error reading bitcode: " << ec.message() << "\n";
    return -1;
  }

  std::unique_ptr<ExecutionEngine> EE(EngineBuilder(*M).create());
  Function *SumFn = (*M)->getFunction("sum");

  std::vector<GenericValue> FnArgs(2);
  FnArgs[0].IntVal = APInt(32,4);
  FnArgs[1].IntVal = APInt(32,5);
  GenericValue Res = EE->runFunction(SumFn, FnArgs);
  outs() << "Sum result: " << Res.IntVal << "\n";

  FnArgs[0].IntVal = Res.IntVal;
  FnArgs[1].IntVal = APInt(32,6);
  Res = EE->runFunction(SumFn, FnArgs);
  outs() << "Sum result: " << Res.IntVal << "\n";

  EE->freeMachineCodeForFunction(SumFn);
  llvm_shutdown();
  return 0;
}
```

In the new MCJIT code example, we again apply the same changes. Check out the following new code, with changes highlighted:

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/ExecutionEngine/JIT.h"
#include "llvm/ExecutionEngine/MCJIT.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/ErrorOr.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/TargetSelect.h"

using namespace llvm;

bool UseMCJIT = true;

int main() {
  InitializeNativeTarget();

  if (UseMCJIT) {
    InitializeNativeTargetAsmPrinter();
    InitializeNativeTargetAsmParser();
  }

  LLVMContext Context;
  ErrorOr<std::unique_ptr<MemoryBuffer>> Buffer =
    MemoryBuffer::getFile("./sum.bc");
  if (Buffer.getError()) {
    errs() << "sum.bc not found\n";
    return -1;
  }

  ErrorOr<Module *> M = parseBitcodeFile(Buffer->get(), Context);
  if (std::error_code ec = M.getError()) {
    errs() << "Error reading bitcode: " << ec.message() << "\n";
    return -1;
  }

  std::unique_ptr<ExecutionEngine> EE;
  if (UseMCJIT)
```

```
      EE.reset(EngineBuilder(*M).setUseMCJIT(true).create());
    else
      EE.reset(EngineBuilder(*M).create());

    Function *SumFn = NULL;
    if (!UseMCJIT)
      SumFn = (*M)->getFunction("sum");
    int (*Sum)(int, int) = NULL;
    if (UseMCJIT)
      Sum = (int (*)(int,int))
        EE->getFunctionAddress(std::string("sum"));
    else
      Sum = (int (*)(int,int)) EE->getPointerToFunction(SumFn);
    int res = Sum(4,5);
    outs() << "Sum result: " << res << "\n";
    res = Sum(res, 6);
    outs() << "Sum result: " << res << "\n";

    if (!UseMCJIT)
      EE->freeMachineCodeForFunction(SumFn);

    llvm_shutdown();
    return 0;
  }
```

Updates for chapter 9

In *Chapter 9, The Clang Static Analyzer*, we have two minor updates, one for the command line and another for the custom checker code.

Generating graphical reports in HTML

The newer Clang Static Analyzer 3.5 no longer generates an HTML report by default if you specify a folder in the `-o` flag. You should explicitly request an HTML report with the `-analyzer-output` flag. The updated command line to generate an HTML report out of Joe's code is the following:

```
$ clang --analyze -Xanalyzer -analyzer-checker=core -Xanalyzer -analyzer-output=html joe.c -o report
```

This command will put the HTML report file inside a folder named `report`.

Remember that you can also use `scan-build` and `scan-view` to see an HTML report. However, notice that `scan-build` and `scan-view` are not installed by default. You should copy them from the `llvm/tools/clang/tools/scan-view` and `llvm/tools/clang/tools/scan-build` source folders to a folder that is in your path.

Extending the static analyzer with your own checkers

In Clang 3.5, we had a minor interface change that was already discussed in *Chapter 9, The Clang Static Analyzer*, since this change was already committed at the time of writing the chapter. We reproduce the code here, highlighting the changes:

```
#include "ClangSACheckers.h"
#include "clang/StaticAnalyzer/Core/BugReporter/BugType.h"
#include "clang/StaticAnalyzer/Core/Checker.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CallEvent.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
using namespace clang;
using namespace ento;

namespace {
class ReactorState {
private:
  enum Kind {Unknown, On, Off} K;
 public:
  ReactorState(unsigned InK): K((Kind) InK) {}
  bool isOn() const { return K == On; }
  bool isOff() const { return K == Off; }
  static unsigned getOn() { return (unsigned) On; }
  static unsigned getOff() { return (unsigned) Off; }
  bool operator==(const ReactorState &X) const {
    return K == X.K;
  }
  void Profile(llvm::FoldingSetNodeID &ID) const {
    ID.AddInteger(K);
  }
};

class ReactorChecker : public Checker<check::PostCall> {
   mutable IdentifierInfo *IIturnReactorOn, *IISCRAM;
   std::unique_ptr<BugType> DoubleSCRAMBugType;
   std::unique_ptr<BugType> DoubleONBugType;
   void initIdentifierInfo(ASTContext &Ctx) const;
```

```
    void reportDoubleSCRAM(const CallEvent &Call,
                           CheckerContext &C) const;
    void reportDoubleON(const CallEvent &Call,
                        CheckerContext &C) const;
  public:
    ReactorChecker();
    /// Process turnReactorOn and SCRAM
    void checkPostCall(const CallEvent &Call, CheckerContext &C) const;
  };
}

REGISTER_MAP_WITH_PROGRAMSTATE(RS, int, ReactorState)

ReactorChecker::ReactorChecker() : IIturnReactorOn(0), IISCRAM(0) {
  // Initialize the bug types.
  DoubleSCRAMBugType.reset(new BugType(this, "Double SCRAM",
                                       "Nuclear Reactor API Error"));
  DoubleONBugType.reset(new BugType(this, "Double ON",
                                    "Nuclear Reactor API Error"));
}

void ReactorChecker::initIdentifierInfo(ASTContext &Ctx) const {
  if (IIturnReactorOn)
    return;
  IIturnReactorOn = &Ctx.Idents.get("turnReactorOn");
  IISCRAM = &Ctx.Idents.get("SCRAM");
}

void ReactorChecker::checkPostCall(const CallEvent &Call,
                                   CheckerContext &C) const {
  initIdentifierInfo(C.getASTContext());
  if (!Call.isGlobalCFunction())
    return;
  if (Call.getCalleeIdentifier() == IIturnReactorOn) {
    ProgramStateRef State = C.getState();
    const ReactorState *S = State->get<RS>(1);
    if (S && S->isOn()) {
      reportDoubleON(Call, C);
      return;
    }
    State = State->set<RS>(1, ReactorState::getOn());
    C.addTransition(State);
    return;
  }
```

```
    if (Call.getCalleeIdentifier() == IISCRAM) {
      ProgramStateRef State = C.getState();
      const ReactorState *S = State->get<RS>(1);
      if (S && S->isOff()) {
        reportDoubleSCRAM(Call, C);
        return;
      }
      State = State->set<RS>(1, ReactorState::getOff());
      C.addTransition(State);
      return;
    }
  }

  void ReactorChecker::reportDoubleON(const CallEvent &Call,
                                      CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
      return;
    BugReport *R = new BugReport(*DoubleONBugType,
                                 "Turned on the reactor two times",
  ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
  }

  void ReactorChecker::reportDoubleSCRAM(const CallEvent &Call,
                                         CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
      return;
    BugReport *R = new BugReport(*DoubleSCRAMBugType,
                                 "Called a SCRAM procedure twice",
  ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
  }

  void ento::registerReactorChecker(CheckerManager &mgr) {
    mgr.registerChecker<ReactorChecker>();
  }
```

Updates for chapter 10

We finish this chapter with the updates for our last chapter. First, we need to update the Makefile used to build our custom refactoring tool from within the LLVM source tree—a library that was renamed in LLVM 3.5 from `libclangRewriteCore` to `libclangRewrite`. The Makefile, with changes highlighted, is as follows:

```
CLANG_LEVEL := ../../..

TOOLNAME = izzyrefactor

# No plugins, optimize startup time.
TOOL_NO_EXPORTS = 1

include $(CLANG_LEVEL)/../../Makefile.config
LINK_COMPONENTS := $(TARGETS_TO_BUILD) asmparser bitreader support mc
option
USEDLIBS = clangTooling.a clangFrontend.a clangSerialization.a
clangDriver.a \
           clangRewriteFrontend.a clangRewrite.a clangParse.a
clangSema.a \
           clangAnalysis.a clangAST.a clangASTMatchers.a clangEdit.a \
           clangLex.a clangBasic.a

include $(CLANG_LEVEL)/Makefile
```

In this chapter, we also mention that you can use the Makefile from *Chapter 4*, *The Frontend*, to build our tool outside of the LLVM tree. To make this easier, the updated Makefile for out-of-tree builds appears in the following code. We highlight the changes compared to the Makefile from *Chapter 4*, *The Frontend*, to show you exactly where you need to change it:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LDFLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXXFLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPPFLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
CLANGLIBS=\
  -Wl,--start-group\
  -lclang\
  -lclangFrontend\
  -lclangDriver\
  -lclangSerialization\
  -lclangParse\
  -lclangSema\
  -lclangAnalysis\
  -lclangEdit\
  -lclangAST\
  -lclangASTMatchers\
  -lclangLex\
  -lclangBasic\
  -lclangTooling\
  -lclangRewrite\
  -Wl,--end-group
LLVMLIBS=$(shell $(LLVM_CONFIG) --libs)
SYSTEMLIBS=$(shell $(LLVM_CONFIG) --system-libs)

PROJECT=izzyrefactor
PROJECT_OBJECTS=IzzyRefactor.o

default: $(PROJECT)

%.o : $(SRC_DIR)/%.cpp
  @echo Compiling $*.cpp
  $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(PROJECT) : $(PROJECT_OBJECTS)
  @echo Linking $@
  $(QUIET)$(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ $(CLANGLIBS)
$(LLVMLIBS) $(SYSTEMLIBS)

clean::
$(QUIET)rm -f $(PROJECT) $(PROJECT_OBJECTS)
```

Next, we show the updated code for our refactoring tool example. Similar to *Chapter 9, The Clang Static Analyzer*, these modifications were already discussed in *Chapter 10, Clang Tools with LibTooling*, because they refer to commits that were already done at the time of writing the chapter. We just applied them here and reproduced the following code, highlighting the changes. Notice that we removed the inclusion of the `OwningPtr.h` header file. The full code is as follows:

```
#include "llvm/Support/CommandLine.h"
#include "clang/Tooling/CompilationDatabase.h"
#include "llvm/Support/ErrorHandling.h"
```

```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Tooling/Refactoring.h"

using namespace clang;
using namespace llvm;
using namespace std;
using namespace clang::ast_matchers;
using clang::tooling::RefactoringTool;
using clang::tooling::Replacement;
using clang::tooling::CompilationDatabase;
using clang::tooling::newFrontendActionFactory;

namespace {
cl::opt<string> BuildPath(
  cl::Positional,
  cl::desc("<build-path>"));
cl::list<string> SourcePaths(
  cl::Positional,
  cl::desc("<source0> [... <sourceN>]"),
  cl::OneOrMore);
cl::opt<string> OriginalMethodName("method", cl::ValueRequired,
  cl::desc("Method name to replace"));
cl::opt<string> ClassName("class", cl::ValueRequired,
  cl::desc("Name of the class that has this method"),
  cl::ValueRequired);
cl::opt<string> NewMethodName("newname",
  cl::desc("New method name"),
  cl::ValueRequired);


class ChangeMemberDecl : public ast_matchers::MatchFinder::MatchCallb
ack{
  tooling::Replacements *Replace;
public:
  ChangeMemberDecl(tooling::Replacements *Replace) : Replace(Replace)
{}
  virtual void run(const ast_matchers::MatchFinder::MatchResult
&Result) {
    const CXXMethodDecl *method =
      Result.Nodes.getNodeAs<CXXMethodDecl>("methodDecl");
    Replace->insert(Replacement(
      *Result.SourceManager,
        CharSourceRange::getTokenRange(
          SourceRange(method->getLocation())), NewMethodName));
  }
};

class ChangeMemberCall : public ast_matchers::MatchFinder::MatchCallb
ack{
  tooling::Replacements *Replace;
public:
  ChangeMemberCall(tooling::Replacements *Replace) : Replace(Replace)
{}
  virtual void run(const ast_matchers::MatchFinder::MatchResult
&Result) {
    const MemberExpr *member =
      Result.Nodes.getNodeAs<MemberExpr>("member");
    Replace->insert(Replacement(
      *Result.SourceManager,
      CharSourceRange::getTokenRange(
        SourceRange(member->getMemberLoc())), NewMethodName));
  }
};
}

int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv);
  string ErrorMessage;
  std::unique_ptr<CompilationDatabase> Compilations (
    CompilationDatabase::loadFromDirectory(
      BuildPath, ErrorMessage));
  if (!Compilations)
    report_fatal_error(ErrorMessage);
  RefactoringTool Tool(*Compilations, SourcePaths);
  ast_matchers::MatchFinder Finder;
  ChangeMemberDecl Callback1(&Tool.getReplacements());
  ChangeMemberCall Callback2(&Tool.getReplacements());
  Finder.addMatcher(
    recordDecl(
      allOf(hasMethod(id("methodDecl",
                       methodDecl(hasName(OriginalMethodName)))),
        isSameOrDerivedFrom(hasName(ClassName)))),
    &Callback1);
  Finder.addMatcher(
    memberCallExpr(
      callee(id("member",
                memberExpr(member(hasName(OriginalMethodName))))),
      thisPointerType(recordDecl(
        isSameOrDerivedFrom(hasName(ClassName))))),
    &Callback2);
  return Tool.runAndSave(newFrontendActionFactory(&Finder).get());
}
```

Summary

In this chapter, we presented modifications to all of our code examples and command-line sequences that you need to consider if you want to exercise this book with the newer LLVM 3.5 release. We also discussed high-level design goals of the LLVM project that affected these changes. Remember that if you want to keep your code updated all the time, it is good practice to use the LLVM code from the SVN trunk and, if you want to understand the motivation behind interface changes, check the SVN commit message history. If you want to read about high-level changes in the LLVM project as they happen, remember to follow the blog page at `http://blog.llvm.org`.