LLVM-C Tutorial and Pharo Bindings

Quentin Ducasse

March 13, 2024

Copyright 2017 by Quentin Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

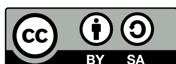
Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations **iii**

I LLVM-C: Tutorial

1 Introduction **3**
1.1 Objectives and scope of the tutorial 3
1.2 Installation 4

2 First function in LLVM IR **5**
2.1 Presentation 5
2.2 Base LLVM components 5
2.3 IR Generation 8
2.4 References 10
2.5 Summary 10

3 Machine Code Generation **11**
3.1 `Triples`, `Targets` and `TargetMachines` 11
3.2 Triple settings 12
3.3 Target Initialisation 12
3.4 Target Machine Initialisation 13
3.5 Emission to file 14
3.6 Summary 15

4 Pharo LLVM Bindings **17**
4.1 How can we use LLVM from Pharo? 17
4.2 Development plan 17
4.3 Module definition 18
4.4 Parameters vector creation 19
4.5 Function definition 20
4.6 Basic block creation 22
4.7 Builder Creation and Operations 22
4.8 Summary 24

5 Advanced Pharo Bindings **27**
5.1 Development plan 27
5.2 Memory Buffer and Emission 27
5.3 Target creation 28
5.4 Enumerations definition 30
5.5 Target Machine Creation 32
5.6 Memory Buffer Emission 33
5.7 Summary 35

Illustrations

2-1 LLVM Type Hierarchy 6
2-2 IR Bitcode 9

Part I

LLVM-C: Tutorial

CHAPTER 1

Introduction

1.1 Objectives and scope of the tutorial

The LLVM Project is a collection of compiler and toolchain technologies. Out of this collection can be found numerous sub-projects such as the ”LLVM Core” libraries providing a modern source-and-target independent optimiser along with code-generation support for many different CPUs. ”Clang” is the LLVM-native C/C++/Obj-C compiler which first goal is to deliver fast compiles as well as useful errors and warning messages.

The ”LLVM Core” libraries are built around a well-specified code representation known as the LLVM Intermediate Representation (or IR). This IR can then be used to perform optimisations and generate target-specific machine code. It is therefore possible to use LLVM as an optimiser and code generator for any programming language.

In the scope of Pharo, LLVM could be used as an optimiser or machine code generator but we have to understand how to perform a translation as an abstract interpretation of the Pharo VM instructions.

Starting from the beginning, the objectives of the following tutorial are the following:

- How to generate LLVM Intermediate Representation?
- How to use this Intermediate Representation to generate machine code?

One important note is that the ”LLVM Core” libraries are written in C++ and since it is not much of use from a Pharo point of view, we will use the ”LLVM C bindings”.

1.2 Installation

In order to install LLVM on your personal computer, several options are possible. It is possible to use the source code and build the LLVM libraries yourself. In our case, the easiest way is to use `brew` and the installation it provides. It can be done using the following command:

```
$ brew install llvm
```

You will now need to add the paths to your `PATH` adding the following lines to your `.bash_profile`:

```
# LLVM
# ====
# PATHS
export PATH="/usr/local/opt/llvm/bin:$PATH"
export
    DYLD_LIBRARY_PATH=/usr/local/Cellar/llvm/9.0.0_1/lib:$DYLD_LIBRARY_PATH
# FLAGS
export LDFLAGS="-L/usr/local/opt/llvm/lib"
export CPPFLAGS="-I/usr/local/opt/llvm/include"
export LDFLAGS="-L/usr/local/opt/llvm/lib
    -Wl,-rpath,/usr/local/opt/llvm/lib"
```

If you have XCode installed on your computer, chances are you might need to switch your XCode command line tools`If you have XCode installed on your computer, chances are you might need to switch your XCode command line toolslanguage=bash $ sudo xcode-select --switch /Library/Developer/CommandLineTools/If you have XCode installed on your computer, chances are you might need to switch your XCode command line tools ```language=bash $ sudo xcode-select --switch /Library/Developer/CommandLineTools/ `` You should now have a working LLVM installation!

CHAPTER 2

First function in LLVM IR

2.1 Presentation

In this chapter, we will be using the LLVM C bindings to build an in-memory representation of an extremely simple function. In order to do so, we will work on a `sum` function whose C equivalent would be:

```
int sum(int a, int b) {
    return a + b;
}
```

To use the LLVM C bindings, we will define the succession of functions to be used in a C program. We will perform in a manual way what an actual compiler could do to an Abstract Syntax Tree result of the function.

2.2 Base LLVM components

Module

The top-level structure in an LLVM program is a module. The official documentation describes this element as a translation unit or a collection of translation units merged together which basically means the module will keep track of functions, global variables, external references and symbols. They are the top-level container for all things defined in LLVM. We can create one by using the `LLVMModuleCreateWithName()` function as follows:

```
LLVMModuleRef mod = LLVMModuleCreateWithName("llvm_c_tutorial");
```

Types

In order to create the `sum` function and add it to the module, we need the following components:

- its return type
- its parameters type (vector)
- a set of basic blocks

Let's first look at the prototype of the function (return and parameters type). Those can be defined in LLVM using the `LLVMTypeRef` holding all the types of the LLVM IR.

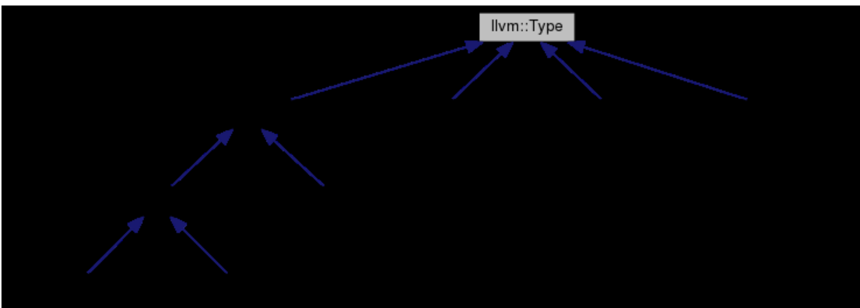


Figure 2-1 LLVM Type Hierarchy

Using those types, we can now build our function prototype. Starting with the parameters type, we can define an `LLVMTypeRef` vector with two `int32` instances created using `LLVMInt32Type()`:

```
LLVMTypeRef param_types[] = { LLVMInt32Type(), LLVMInt32Type() };
```

Note that all the types in LLVM can be used with the LLVM<type>Type() constructor. This is the case with our two `LLVMInt32Type()`. Next comes the function type that is defined as follows:

```
LLVMTypeRef ret_type = LLVMFunctionType(LLVMInt32Type(),
    param_types, 2, 0);
```

The `LLVMFunctionType()` constructor takes for arguments:

1. the function's return type
2. the function's parameters type vector
3. the function's arity
4. a boolean telling if the function is variadic or not (accepts a variable number of arguments)

Finally, we can add the function we just defined to the module we defined earlier, give it a name and get an `LLVMValueRef` as a result. This reference is the concrete location of the function in memory.

Basic Blocks

A basic block represents a single entry or exit section of code, it contains a list of instructions and it belongs to a function. Those blocks only have an entry and exit point, meaning the list of instructions will have to be executed from the beginning to the end. This means there cannot be conditional loops or jumps of any kind inside those basic blocks. We can define our basic block with the `LLVMAppendBasicBlock()` function as follows:

```
LLVMBasicBlockRef entry = LLVMAppendBasicBlock(sum, "entry");
```

The `LLVMAppendBasicBlock` links the `entry` basic block to our previously defined function. We now have a module that contains all the references to the function we wanted to define and the different types and basic blocks it needed.

Instruction Builder

In order to add instructions to our function's unique basic block, we need to use an instruction builder. This component, described in the documentation as a point within a basic block and is the exclusive means of building instructions using the C interface, is created using the `LLVMCreateBuilder()` function. Then, in the same way we added the basic block to the function, we position the builder to start in the entry of the basic block using the `LLVMPositionBuilderAtEnd()` function.

```
LLVMBuilderRef builder = LLVMCreateBuilder();
LLVMPositionBuilderAtEnd(builder, entry);
```

Now comes the function itself. We need to add the two arguments of our function and return the result. In order to do this, we first have to retrieve the parameters passed to the function. This can be done with the function `LLVMGetParam()` that takes a reference to the function and the index of the desired parameter. Then, we need to properly create an instruction to sum them up using `LLVMBuildAdd()` function, and providing a return holder (`tmp` here). `LLVMBuildAdd` takes the two integers to add and a name to give to the result. Note that this name is required due to LLVM's policy saying that all instructions have to produce intermediate results

```
LLVMValueRef tmp = LLVMBuildAdd(builder, LLVMGetParam(sum, 0),
    LLVMGetParam(sum, 1), "tmp");
```

Finally, we call the `LLVMBuildRet()` function to generate the return statement and arrange for the temporary result.

```
LLVMBuildRet(builder, tmp);
```

Analysis

Our module is complete and we can check for any errors or exceptions by using the tools present in the `analysis` library. For example, we can verify our module using the `LLVMVerifyModule` function. This function will analyse our module and report for any errors:

```
char *error = NULL;
LLVMVerifyModule(mod, LLVMAbortProcessAction, &error);
LLVMDisposeMessage(error);
```

2.3 IR Generation

The LLVM IR is a strictly defined language that can be qualified as a midway point between assembly and C. This LLVM IR can take three different forms and aims at platform portability. Those three forms are the following:

- an in-memory set of objects, which we are using here
- a textual language like assembly
- a string of bytes binary encoded, called bitcode

We are currently using the first form but we can get the two others. Let's start by writing the bitcode of the module to a file. In order to do so, we will use the`bitwriter` library and its `LLVMWriteBitcodeToFile` function:

```
if (LLVMWriteBitcodeToFile(mod, "sum.bc") != 0) {
    fprintf(stderr, "error writing bitcode to file, skipping\n");
}
```

In order to get this bitcode, we need to compile the C program as specified in the makefile. If we look closer at the different commands we are executing, we can find the following:

```
$ clang `llvm-config --cflags` -c sum.c
```

The first command compiles the C source code into an object file `sum.o`. We need to provide the `--cflags` in order for LLVM to work correctly. The next command will call the linker:

```
$ clang++ `llvm-config --cxxflags --ldflags --libs core analysis
    native bitwriter --system-libs` sum.o -o sum
```

Even though we are writing a C program, please remember that we are only using the LLVM C bindings and API while LLVM itself is written in C++. That is why we need the C++ linker in order for the generation of the executable to work.

Now in order to execute the output, simply perform:

```
$ ./sum
```

The bitcode will then be written to the file `sum.bc`. This bitcode is the second form of the IR.


```
4243C0DE 35140000 05000000 620C3024 4959BE66 EFD33E2D 44013205 00000000 210C0000
DF000000 0B022100 02000000 16000000 07812391 41C80449 06103239 9201840C 25050819
1E048B62 800C4502 42920B42 64103214 3808184B 0A323288 4870C421 23441287 8C104192
0264C808 B1142043 468820C9 01323284 182A282A 90317CB0 5C9120C3 C8000000 89200000
0B000000 3222C808 20648504 9321A484 049321E3 84A19014 124C868C 0B84644C 10147304
60500600 94818011 00000000 1A210C99 30D8CF38 03B51043 2A600800 00000000 00000000
00000006 40628340 D1710000 80580C00 89000000 3308801C C4E11C66 14013D88 433884C3
8C428007 79780773 98710CE6 000FED10 0EF4800E 330C421E C2C11DCE A11C6630 053D8843
3884831B CC033DC8 433D8C03 3DCC788C 7470077B 08077948 87707007 7A700376 78877020
8719CC11 0EEC900E E1300F6E 300FE3F0 0EF0500E 3310C41D DE211CD8 211DC261 1E663089
3BBC833B D04339B4 033CBC83 3C84033B CCF01476 60077B68 07376887 72680737 80877090
87706007 76280776 F8057678 87778087 5F088771 18877298 87799881 2CEEF00E EEE00EF5
C00EEC30 0362C8A1 1CE4A11C CCA11CE4 A11CDC61 1CCA211C C4811DCA 6106D690 4339C843
39984339 C84339B8 C3389443 3888033B 94C32FBC 833CFC82 3BD4033B B0C30CC7 69877058
87727083 74680778 60877418 8774A087 19CE530F EE000FF2 500EE490 0EE3400F E1200EEC
500E3320 281DDCC1 1EC2411E D2211CDC 811EDCE0 1CE4E11D EA011E66 185138B0 433A9C83
3BCC5024 7660077B 68073760 87777807 7898514C F4900FF0 500E331E 6A1ECA61 1CE8211D
DEC11D7E 011EE4A1 1CCC211D F0610654 858338CC C33BB043 3DD04339 FCC23CE4 433B88C3
3BB0C38C C50A8779 98877718 87740807 7A280772 98815CE3 100EECC0 0EE5500E F33023C1
D2411EE4 E117D8E1 1DDE011E 6648193B B0833DB4 831B84C3 388C4339 CCC33CB8 C139C8C3
3BD4033C CC48B471 08077660 07710887 71588719 DBC60EEC 600FEDE0 06F0200F E5300FE5
200FF650 0E6E100E E3300EE5 300FF3E0 06E9E00E E4500EF8 00000000 A9180000 0B000000
0B0A7228 87778007 7A587098 433DB8C3 38B04339 D0C382E6 1CC6A10D E8411EC2 C11DE621
1DE8211D DEC11D00 D1100000 06000000 07CC3CA4 833B9C03 3B94033D A0833C94 433890C3
01000000 61200000 08000000 1304C109 016C100E 04000000 03000000 36304CCC 730005D1
4C110600 00000000 71200000 03000000 320E1022 8400D901 00000000 00000000 5D0C0000
07000000 12039411 73756D39 2E302E30 6D795F6D 6F64756C 65000000 00000000
```


Figure 2-2 IR Bitcode

We can finally get the text output by using one of the LLVM tools, the disassembler. This tool allows disassembling bitcode and outputting the IR textual representation. It can be done by using `llvm-dis` as follows:

```
$ llvm-dis sum.bc
```

If we now look at the output `sum.ll`, we can see:

```
; ModuleID = 'sum.bc'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"

define i32 @sum(i32, i32) {
entry:
  %tmp = add i32 %0, %1
  ret i32 %tmp
}
```

This is the LLVM representation of our `sum` function! Note that all the different steps of the generation can be performed through the use of the makefile by doing:

```
$ make sum.o
$ make sum
$ make sum.bc
$ make sum.ll
```

Note that `make all` is the equivalent of `make sum`.

2.4 References

The following links are useful pointers to the official LLVM reference API documentation we used in this chapter:

- Moduleshttps://llvm.org/doxygen/group__LLVMCCoreModule.html*[Modules](https://llvm.org/doxygen/group*LLVMCCoreModule.html)
- Function Typeshttps://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html*[Function Types](https://llvm.org/doxygen/group*LLVMCCoreTypeFunction.html)
- Basic Blockshttps://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html*[Basic Blocks](https://llvm.org/doxygen/group*LLVMCCoreValueBasicBlock.html)
- Instruction Builderhttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html*[Instruction Builder](https://llvm.org/doxygen/group*LLVMCCoreInstructionBuilder.html)
- Int Typeshttps://llvm.org/doxygen/group__LLVMCCoreTypeInt.html*[Int Types](https://llvm.org/doxygen/group*LLVMCCoreTypeInt.html)
- Analyseshttps://llvm.org/doxygen/group__LLVMCAnalysis.html*[Analyses](https://llvm.org/doxygen/group*LLVMCAnalysis.html)
- Bit Writerhttps://llvm.org/doxygen/group__LLVMCBitWriter.html*[Bit Writer](https://llvm.org/doxygen/group*LLVMCBitWriter.html)

2.5 Summary

In this chapter we learned how to build a function using the LLVM IR and create the bitcode related to this IR. We explored the different types and components that the C API provides and we created the function and its basic block in a manual way. Ideally, what we just did will be scripted.

CHAPTER 3

Machine Code Generation

3.1 Triples, Targets and TargetMachines

In the last chapter we generated the bitcode for the host machine. How can we specify the machine we want to generate code for (and by code I mean machine code)? This idea would use the main strength of LLVM, with one IR we can generate any specified machine code. Now that we have the IR ready by creating it last chapter, how can we specify the architecture we want to deploy it on?

In order to answer this issue, LLVM uses a `Target`/`TargetMachine` double representation of any architecture and a `Triple` as a textual descriptor of the architecture, vendor and operating system.

Basically, `Triples` are helpers for configuration. Their names come from the fact that they used to contain only three fields and are strings in the canonical form of: ARCHITECTURE - VENDOR - OPERATING SYSTEM (- ENVIRONMENT)

If some of the fields are not specified, the default ones will be used. The list of all the architectures, vendors, operating systems and environments can be found under: Triplehttps://llvm.org/doxygen/classllvm_1_1Triple.html#details*11*Triple.html#details).

The `Target` on the other hand is an object used only by LLVM under the hood in order to profile the target you will be using and be used by the `TargetMachine` object later on. These `Target` objects need to be initialised correctly before being deduced from a `Triple`. They are used to describe that for which code may be intended. They are used both during the lowering of LLVM IR to machine code and by some optimisations passes.

The `TargetMachine` on the other hand takes a reference to a `Target` and then uses a description of the machine through the specification of the triple and the cpu. Last things provided to the `TargetMachine` are indicators on optimisations or the type of model we want. Fortunately, those last bits are elements taken from enumerations and will not be the source of any issue.

3.2 Triple settings

The triple can either be obtained automatically from your actual machine by using`LLVMGetDefaultTargetTriple()` or can be specified following the form specified earlier. In our example, using `LLVMGetDefault-TargetTriple()` can be done as follows:

```
char* triple = LLVMGetDefaultTargetTriple(); // Using the triple of
    my machine
printf("%s\n",triple);
>>> x86_64-apple-darwin18.7.0
```

In order to get a coherent output, we will use a provided triple from now on:

```
char triple[] = "x86_64";
char cpu[] = "";
```

The `cpu` string is used later on but can be defined by default by LLVM so we will be using it this way. Basically, it will guess it based on the first part of the triple string.

3.3 Target Initialisation

Before being deduced from a `Triple`, any `Target` has to be initialised. This initialisation can be specified case by case by using the different libraries but since we want to be able to create machine code for any supported machine, we need to use all the possible initialisations and use them all. These initialisations are the four following:

```
LLVMInitializeAllTargets();
LLVMInitializeAllTargetMCs();
LLVMInitializeAllTargetInfos();
LLVMInitializeAllAsmPrinters();
```

- LLVMInitializeAllTargets()https://llvm.org/doxygen/group__LLVMCTarget.html#gace43bdd15ad030e38c41ae1cb43a *tializeAllTargets()](https://llvm.org/doxygen/group*LLVMCTarget.html#gace43bdd15ad030e38c41ae1cb43a2539)
- LLVMInitializeAllTargetMCs()https://llvm.org/doxygen/group__LLVMCTarget.html#ga750b05e81ab9a921bb6d2e9b *tializeAllTargetMCs()](https://llvm.org/doxygen/group*LLVMCTarget.html#ga750b05e81ab9a921bb6d2e9b912e66eb)
- LLVMInitializeAllTargetInfos()https://llvm.org/doxygen/group__LLVMCTarget.html#ga40188f383ddf8774ede38e809 *tializeAllTargetInfos()](https://llvm.org/doxygen/group*LLVMCTarget.html#ga40188f383ddf8774ede38e8098da9a9a)
- LLVMInitializeAllAsmPrinters()https://llvm.org/doxygen/group__LLVMCTarget.html#gadcbb41ca7051aca660022e70 *tializeAllAsmPrinters()](https://llvm.org/doxygen/group*LLVMCTarget.html#gadcbb41ca7051aca660022e70ee62dd7f

1. The first initialisation `LLVMInitializeAllTargets()` enables the program to access all available targets that LLVM is configured to support.
2. The second initialisation `LLVMInitializeAllTargetMCs()` enables the program to access all available target Machine Codes.
3. The third initialisation `LLVMInitializeAllTargetInfos()` makes the program able to link all available targets that LLVM is configured to support.
4. The last initialisation `LLVMInitializeAllAsmPrinters()` gives access to all of the available ASM printers to the program.

Basically, **1** holds the description, **2** the machine code, **3** the linkers and **4** the ASM printer.

Once the target initialisation is done, we can extract the `LLVMTargetRef` we want from a given triple. This is done by creating an `LLVMTargetRef` that will be filled by the `LLVMGetTarget-FromTriple()` function. Adding an error pointer, we can get the following piece of code:

```
LLVMTargetRef targetRef;
char** errPtrTriple;
LLVMBool resTriple = LLVMGetTargetFromTriple(triple, &targetRef,
    errPtrTriple);
if (resTriple != 0)
{
    printf("%s\n",*errPtrTriple);
}
```

LLVMGetTargetFromTriple()https://llvm.org/doxygen/c_2TargetMachine_8h.html#a7a746a65818e0b6bd86e5f00a568e
*getMachine*8h.html#a7a746a65818e0b6bd86e5f00a568e301)

3.4 Target Machine Initialisation

Now that we have a proper `Target` initialised, the next step is to create a`TargetMachine`. This can be done by using the function `LLVMCreateTargetMachine()`. However, this function takes seven arguments so let's take a closer look at how we are going to use it:

```
LLVMTargetMachineRef targetMachineRef =
    LLVMCreateTargetMachine(targetRef, triple, cpu, "",
    LLVMCodeGenLevelNone, LLVMRelocDefault, LLVMCodeModelDefault);
```

LLVMCreateTargetMachinehttps://llvm.org/doxygen/c_2TargetMachine_8h.html#a9b0b2b1efd30fad999f2b2a7fdbf849
*getMachine*8h.html#a9b0b2b1efd30fad999f2b2a7fdbf8492)

The result is an `LLVMTargetMachineRef`, what we need and were expecting. The seven arguments are the following:

- an `LLVMTargetRef`: The `Target` we defined earlier
- a string `Triple`: The `Triple` representing the target machine
- a string `cpu`: The cpu of the target machine
- a string `features`:
- an `LLVMCodeGenOptLevel`: The level of optimisation we want, this is an enumeration we will have to choose from (`None`,`Less`,`Default` and `Agressive`)
- an `LLVMRelocMode`: The mode of relocation we want, another enumeration we have to choose from (`Default`,`Static`,`PIC`,etc.)
- an `LLVMCodeModel`: This is a term from AMD64 ABIhttps://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf (chapter 3.5) that defines the offset between data and code within instructions, also an enumeration we can choose from (`Default`,`JITDefault`,`Tiny`,`Small`

You can find more details on the existing elements of the enumerations by looking at the following links in the documentation:

- LLVMCodeGenOptLevelhttps://llvm.org/doxygen/c_2TargetMachine_8h.html#acad7f73c0a2e7db5680d80becd5
*getMachine*8h.html#acad7f73c0a2e7db5680d80becd5e719b)
- LLVMRelocModehttps://llvm.org/doxygen/c_2TargetMachine_8h.html#ac6ed8c89bb69e7a56474cac6cf0ffb672*T*
*getMachine*8h.html#ac6ed8c89bb69e7a56474cac6cf0ffb67)
- LLVMCodeModelhttps://llvm.org/doxygen/c_2TargetMachine_8h.html#a333ec2da299d964c0885bee025bef68c2
*getMachine*8h.html#a333ec2da299d964c0885bee025bef68c)

3.5 Emission to file

Finally, once the `TargetMachine` is correctly created, we can emit the module either to an object or an ASM file. This is done through the function `LLVMTargetMachineEmitToFile()`.

The function takes several arguments, from which:

- the `TargetMachine` reference,
- the module,
- a name for the new file (we specified `.o` here for the object file),
- the type of file, either `LLVMObjectFile` or `LLVMAssemblyFile` (both are taken from the `LLVMCodeGenFileType` enumeration)
- an error pointer

In the end, the result looks like:

```
char** errPtrFileObj;
LLVMBool resFileObj = LLVMTargetMachineEmitToFile(targetMachineRef,
    mod, "sum_llvm.o", LLVMObjectFile, errPtrFileObj);
if (resFileObj != 0)
{
    printf("%s\n",*errPtrFileObj);
}
```

or

```
char** errPtrFileAsm;
LLVMBool resFileAsm = LLVMTargetMachineEmitToFile(targetMachineRef,
    mod, "sum_llvm.asm", LLVMAssemblyFile, errPtrFileAsm);
if (resFileAsm != 0)
{
    printf("%s\n",*errPtrFileAsm);
}
```

In order to get the output, run the provided makefile. It differs from the first one as it needs all the libraries to run the target initialisation. The output will generate a `.o` and `.asm` file. If we take a look at the `.asm` file we can see:

```
.section  __TEXT,__text,regular,pure_instructions
.macosx_version_min 10, 14
.globl  _sum
.p2align  4, 0x90
_sum:
.cfi_startproc
addl  %esi, %edi
movl  %edi, %eax
retq
.cfi_endproc


.subsections_via_symbols
```

3.6 Summary

What did we learn in this chapter? We extended the function we defined in the previous chapter by now being able to state and select a specific architecture to target and aim for when building the machine code. Those steps lead to a file (either object or ASM). When creating the Pharo bindings corresponding to the LLVM-C interface, we will want to emit the machine code to a memory buffer rather than a file and this can be done the exact same way we did while emitting to a file but using the function`LLVMTargetMachineEmitToMemoryBuffer()`

CHAPTER 4

Pharo LLVM Bindings

4.1 How can we use LLVM from Pharo?

In order to use LLVM from Pharo, the main solution is to call the LLVM C bindings and use those with the help of the uFFI package (Unified Foreign Function Interface). This package is a façade to all external libraries that one might want to use from within Pharo. C libraries can therefore be called and used.

Even though Pharo and C are extremely different, the calls can operate by marshalling correctly the different types, structures and by binding the different functions with respect to some rules. In order to use the Pharo uFFI library, we will go over the piece of code we wrote in the first chapter and try to make it work from within a Pharo image.

4.2 Development plan

The uFFI library is best used either as a façade design pattern where one object will hold all of the different bindings that are present in the library we want to use. This way works well if the library you are writing a binding for is small and can be contained within a few functions. Another way to better transcribe your library that is more suitable for important libraries is to replicate an object-oriented behaviour for the different structures or objects you are dealing with. As the LLVM library is wide and might keep on expanding, we will choose the second option.

By looking back at the final code from the first chapter, the different steps we will be aiming for are:

- Module definition
- Parameters vector creation
- Function definition
- Basic block creation
- Builder creation and operations
- Output to a memory buffer

4.3 Module definition

The first element and top container of LLVM is a `Module`. Let's define the class within a freshly created package:

```
FFIExternalObject subclass: #LLVMModule
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

The first methods we will want to define in this class (but also in all the other classes using external bindings) are `ffiLibrary`:

```
LLVMModule >> ffiLibrary
  ^ self class ffiLibrary
```

```
LLVMModule class >> ffiLibrary
  ^ 'libLLVM.dylib' asFFILibrary
```

Those two methods indicate to the class where to look for the library we want to call. In our case, the `libLLVM.dylib` is the place to go. Note that this name can differ depending on the platform you are using (e.g. `*.so` on Linux or`*.dll` on Windows).

Now, the function we want to use is `LLVMCreateModuleWithName()` (LLVMModuleCreateWithName()https://llvm.org/doxygen/group__LLVMCCoreModule.html#ga8cf6711b9359fb55d081bfc5e664370c*Now, the function we want to use is 'LLVMCreateModuleWithName()' ([LLVMModuleCreateWithName()](https://llvm.org/doxygen/group*LLVMCCoreModule.html#ga8cf6711b9359fb55d081bfc5e664370c)) In order to output an `LLVMModule`, we can define the binding as a class function that could allow us to create a module as follows:

```
LLVMModule withName: 'aModule'.
>>> LLVMModule
```

This can be done by defining the binding on the class side of `LLVMModule`:

```
LLVMModule class >> withName: aName
  ^ self ffiCall: #(LLVMModule LLVMModuleCreateWithName(String
     aName))
```

Now that our module creation is working, we can add some other functions that are used on modules from within LLVM. Note that this can be done easily due to the choice of the bindings structure we made. Adding a function while using the façade design pattern would make us take a reference to the module as well. Here, we can just use any function that needs a module as an argument and pass our object to it! For example, the function `LLVMGetTarget()` (LLVMGetTargethttps://llvm.org/doxygen/group__LLVMCCoreModule.html#ga25a0be3489b6c0f0b517828d84e376f3*Now that our module creation is working, we can add some other functions that are used on modules from within LLVM. Note that this can be done easily due to the choice of the bindings structure we made. Adding a function while using the façade design pattern would make us take a reference to the module as well. Here, we can just use any function that needs a module as an argument and pass our object to it! For example, the function 'LLVMGetTarget()' ([LLVMGetTarget](https://llvm.org/doxygen/group*LLVMCCoreModule.html#ga25a0be3489b6c0f0b517828d84e376f3)) that outputs the `Triple` representing the target of a given module and which signature is:

```
const char* LLVMGetTarget ( LLVMModuleRef   M )
```

Can be written in Pharo as:

```
LLVMModule >> target
  self ffiCall: #(const char *LLVMGetTarget(LLVMModule self))
```

A last thing we need to do for our module to be fully operational (for now) is to write its `dispose` function. In a C world, this would be required when the object is no longer in use (in a way `free` would be used). In Pharo, we can encapsulate this behaviour in the `finalize` method that is called automatically while garbage collecting a Pharo object. We would like to call `LLVMDisposeModule*()` (LLVMDiaposeModulehttps://llvm.org/doxygen/group__LLVMCCoreModule.html#ga4acadb366d5ff0405371508ca741a5bf*A last thing we need to do for our module to be fully operational (for now) is to write its ‘dispose‘ function. In a C world, this would be required when the object is no longer in use (in a way ‘free‘ would be used). In Pharo, we can encapsulate this behaviour in the ‘finalize‘ method that is called automatically while garbage collecting a Pharo object. We would like to call ‘LLVMDisposeModule*()‘ ([LLVMDiaposeModule](https://llvm.org/doxygen/group*LLVMCCoreModule.html#ga4acadb366d5ff04053715
during this phase. This can be done by defining the two methods:

```
LLVMModule >> finalize
  self dispose
```

```
LLVMModule >> dispose
  self ffiCall: #(void LLVMDisposeModule(LLVMModule self))
```

4.4 Parameters vector creation

Now that our module is done, we need to get the parameters array. It consists of the types of the signature of a function. In our example:

```
LLVMTypeRef param_types[] = { LLVMInt32Type(), LLVMInt32Type() };
```

In order to replicate this behaviour we need to define what an `LLVMInt32Type` is and to see how to create an external array in Pharo. First, we can define a superclass `LLVMType` from which our `LLVMInt32` will inherit. This is done as follows:

```
FFIExternalObject subclass: #LLVMType
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMType subclass: #LLVMInt32
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

Do not forget to define the `ffiLibrary` class and instance methods under`LLVMType` as it is needed in any class using bindings! Now, to create a proper`LLVMInt32`, we can override the `new` method as follows:

```
LLVMInt32 class >> new
  ^ self ffiCall: #(LLVMInt32 LLVMInt32Type(void))
```

In order to create an array and populate it with the correct type, we need to use the uFFI featured object `FFIArray`. It can be defined and created as follows:

```
FFIArray subclass: #LLVMParameterArray
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMParameterArray class >> withSize: aNumber
  ^  FFIArray newType: LLVMType size: aNumber
```

In the end, our array can be created and populated as shown below:

```
paramArray := LLVMParameterArray withSize: 2.
paramArray at: 1 put: (LLVMInt32 new handle getHandle).
paramArray at: 2 put: (LLVMInt32 new handle getHandle).
```

4.5 Function definition

An important aspect of the functions as they are defined in LLVM is that they are both represented under a `Type` and a `Value`. The `Type` represents the function signature and adding the signature to a module will output a `Value`. This `Value` is the location of the function in memory. `Values` are used in LLVM to represent many different objects such as function parameters, instructions or basically any object that has an in-memory representation.

First things first, let's define a new `LLVMType`, `LLVMFunctionSignature`:

```
LLVMType subclass: #LLVMFunctionSignature
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

From within this class, the method that can instantiate it is `LLVMFunctionType()` (LLVMFunctionTypehttps://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html#ga8b0c32e7322e5c6c1bf7eb95b0961707*From within this class, the method that can instantiate it is 'LLVMFunctionType()' ([LLVMFunctionType](https://llvm.org/doxygen/group*LLVMCCoreTypeFunction.html#ga8b0c32e7322e5c6c1bf7eb95b0961707))
that we can bind as follows:

```
LLVMFunctionSignature class >> withReturnType: aType
    parametersVector: anArray arity: anInteger andIsVaridic: aBoolean
^ self ffiCall: #(#LLVMFunctionSignature LLVMFunctionType(LLVMType
    aType,
                                                LLVMParameterArray anArray,
                                                int anInteger,
                                                Boolean aBoolean))
```

Now, we can populate the instance side with some accessors that might come in handy:

```
LLVMFunctionSignature >> returnType
  ^ self ffiCall: #(LLVMType LLVMGetReturnType(LLVMFunctionSignature
    self))
```

LLVMGetReturnType()https://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html#gacfa4594cbff421733add602a413ca
*tReturnType()](https://llvm.org/doxygen/group*LLVMCCoreTypeFunction.html#gacfa4594cbff421733add602a413cae9f)

```
LLVMFunctionSignature >> parametersNumber
  ^ self ffiCall: #(uint LLVMCountParamTypes(LLVMFunctionSignature
    self))
```

LLVMCountParamTypes()https://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html#ga44fa41d22ed1f589b8202272f5
*CountParamTypes()](https://llvm.org/doxygen/group*LLVMCCoreTypeFunction.html#ga44fa41d22ed1f589b8202272f54aad7

```
LLVMFunctionSignature >> isVaridic
  ^ self ffiCall: #(Boolean
    LLVMIsFunctionVarArg(LLVMFunctionSignature self))
```

LLVMIsFunctionVarArg()https://llvm.org/doxygen/group__LLVMCCoreTypeFunction.html#ga2970f0f4d9ee8a0f811f762
*IsFunctionVarArg()](https://llvm.org/doxygen/group*LLVMCCoreTypeFunction.html#ga2970f0f4d9ee8a0f811f762fb2fa

Finally, we can add the function to the module and get the output `Value`. In order to do so, let's first define an `LLVMValue` and a subclass `LLVMFunction`.

```
FFIExternalObject subclass: #LLVMValue
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMValue subclass: #LLVMFunction
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

Then, the function `LLVMAddFunction()` (LLVMAddFunctionhttps://llvm.org/doxygen/group_
_LLVMCCoreModule.html#ga12f35adec814eb1d3e9a2090b14f74f5*Then, the function 'LLVMAddFunction()' ([LLVMAddFunction](https://llvm.org/doxygen/group*LLVMCCoreModule.html#ga12f35adec814eb1d3e9a2090b1
can be defined in the `LLVMFunctionSignature` as:

```
LLVMFunctionSignature >> addToModule: aModule withName: aName
  ^ self ffiCall: #(LLVMFunction LLVMAddFunction(LLVMModule aModule,
                                                  String aName,


    LLVMFunctionSignature self))
```

Some accessors can be defined as well in the `LLVMFunction` class, from which I chose `LLVMGetFirstParam()`, `LLVMGetLastParam()`, `LLVMGetParam()`, ... that I will showcase below. Those accessors use another subclass of `LLVMValue`, an `LLVMFunctionParameter`.

```
LLVMValue subclass: #LLVMFunctionParameter
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMFunction >> firstParameter
  ^ self ffiCall: #(LLVMFunctionParameter
    LLVMGetFirstParam(LLVMFunction self))
```

```
LLVMFunction >> lastParameter
  ^ self ffiCall: #(LLVMFunctionParameter
    LLVMGetLastParam(LLVMFunction self))
```

```
LLVMFunction >> nextParameterOf: aParameter
  ^ self ffiCall: #(LLVMFunctionParameter
    LLVMGetNextParam(LLVMFunction self))
```

```
LLVMFunction >> parameterAtIndex: anInteger
  ^ self ffiCall: #(LLVMFunctionParameter LLVMGetParam(LLVMFunction
    self, uint anInteger))
```

```
LLVMFunction >> parametersNumber
  ^ self ffiCall: #(uint LLVMCountParams(LLVMFunction self))
```

```
LLVMFunction >> previousParameterOf: aParameter
  ^ self ffiCall: #(LLVMFunctionParameter
    LLVMGetPreviousParam(LLVMFunction self))
```

4.6 Basic block creation

Basic block can be created from a function by appending them to it with the help of the `LLVMAppendBasicBlock()` function. This function will correspond to the way we will be instantiating our `LLVMBasicBlock`.

```
FFIExternalObject subclass: #LLVMBasicBlock
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMBasicBlock >> linkToFunction: aFunctionValue withName: aName
  ^ self ffiCall: #(LLVMBasicBlock LLVMAppendBasicBlock(LLVMValue
    aFunctionValue,
                                                    String
    aName))
```

LLVMAppendBasicBlockhttps://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html#ga74f2ff28344ef72a8206b9c592
*pendBasicBlock](https://llvm.org/doxygen/group*LLVMCCoreValueBasicBlock.html#ga74f2ff28344ef72a8206b9c5925be543)

And we can now add some helper functions such as:

```
LLVMBasicBlock >> name
  ^ self ffiCall: #(const char* LLVMGetBasicBlockName(LLVMBasicBlock
    self))
```

```
LLVMBasicBlock >> parent
  ^ self ffiCall: #(LLVMValue LLVMGetBasicBlockParent(LLVMBasicBlock
    self))
```

```
LLVMBasicBlock >> delete
  self ffiCall: #(void LLVMDeleteBasicBlock(LLVMBasicBlock self))
```

4.7 Builder Creation and Operations

The builder is created through the `LLVMCreateBuilder()` function (LLVMCreateBuilderhttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gaceec9933fd90a94ea5ebb40eedf6136d*The builder is created through the 'LLVMCreateBuilder()' function ([LLVMCreateBuilder](https://llvm.org/-doxygen/group*LLVMCCoreInstructionBuilder.html#gaceec9933fd90a94ea5ebb40eedf6136d)). This function will help us initialise our builder by creating the following class and method:

```
FFIExternalObject subclass: #LLVMBuilder
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

```
LLVMBuilder class >> new
  ^ self ffiCall: #(LLVMBuilder LLVMCreateBuilder(void))
```

A similar call to `finalize` and `dispose` can be done:

```
LLVMBuilder >> dispose
  self ffiCall: #(void LLVMDisposeBuilder(LLVMBuilder self))
```

```
LLVMBuilder >> finalize
  ^ self dispose
```

Now, we need to define the actual builder operations (we will simply define the add operation as well as the building of the return value) and positioning. In order to define the `LLVMBuildAdd` function (LLVMBuildAddhttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#ga5e20ba4e932d72d97a69e07ff54cfa81*Now, we need to define the actual builder operations (we will simply define the add operation as well as the building of the return value) and positioning. In order to define the 'LLVMBuildAdd' function ([LLVMBuildAdd](https://llvm.org/doxygen/group*LLVMCCoreInstructionBuilder.html#
we proceed as follows:

```
LLVMBuilder >> buildAdd: aValue to: anotherValue andStoreUnder:
    aTemporaryValueName
  ^ self ffiCall: #(LLVMValue LLVMBuildAdd(LLVMBuilder self,
                                            LLVMValue    aValue,
                                            LLVMValue
    anotherValue,
                                            const char *
    aTemporaryValueName ))
```

Next, `LLVMBuildRet` (LLVMBuildRethttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gae4c870d69f9787fe98a824a634473155*Next, 'LLVMBuildRet' ([LLVMBuildRet](https://llvm.org/-doxygen/group*LLVMCCoreInstructionBuilder.html#gae4c870d69f9787fe98a824a634473155)):

```
LLVMBuilder >> buildReturnStatementFromValue: aValue
  ^ self ffiCall: #(LLVMValue LLVMBuildRet(LLVMBuilder self,
                                           LLVMValue aValue))
```

Finally, we will need to position our builder at the end of a basic block. We will define some more positioning functions as well. The first one, and the one we will use is `LLVMPositionBuilderAtEnd()` (LLVMPositionBuilderAtEndhttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gafa58ecb369fc661ff7e58c19c46053f0*Finally, we will need to position our builder at the end of a basic block. We will define some more positioning functions as well. The first one, and the one we will use is 'LLVMPositionBuilderAtEnd()' ([LLVMPositionBuilderAtEnd](https://llvm.org/doxygen/-group*LLVMCCoreInstructionBuilder.html#gafa58ecb369fc661ff7e58c19c46053f0)):

```
LLVMBuilder >> positionBuilderAtEndOfBasicBlock: aBasicBlock
  ^ self ffiCall: #(void LLVMPositionBuilderAtEnd(LLVMBuilder self,
                                  LLVMBasicBlock aBasicBlock))
```

Next come `LLVMPositionBuilderBefore()` (LLVMPositionBuilderBeforehttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gaed4e5d25ea3f3a3554f3bc4a4e22155a*Next come 'LLVM-PositionBuilderBefore()' ([LLVMPositionBuilderBefore](https://llvm.org/doxygen/group*LLVMCCoreInstructionBuilder.htm
and `LLVMPositionBuilder()` (LLVMPositionBuilderhttps://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gaf8bc5c4564732d52c5aaae2cf56d2f5c*Next come 'LLVMPositionBuilderBefore()' ([LLVMPosition-BuilderBefore](https://llvm.org/doxygen/group__LLVMCCoreInstructionBuilder.html#gaed4e5d25ea3f3a3554f3bc4a4e22155a)
and 'LLVMPositionBuilder()' ([LLVMPositionBuilder](https://llvm.org/doxygen/group*LLVMCCoreInstructionBuilder.html
Those two functions are used to position the builder on a particular instruction. We can implement them as follows:

```
LLVMBuilder >> positionBuilderBeforeInstruction: anInstruction
  ^ self ffiCall: #(void LLVMPositionBuilderBefore(LLVMBuilder self,
                                                   LLVMInstruction
    anInstruction))
```

```
LLVMBuilder >> positionBuilderOnInstruction: anInstruction
    inBasicBlock: aBasicBlock
  ^ self ffiCall: #(void LLVMPositionBuilder(LLVMBuilder     self,
                                             LLVMBasicBlock
    aBasicBlock,
                                             LLVMInstruction
    anInstruction))
```

The builder is now complete! By adding a small method to the `LLVMModule` to emit the code to a memory buffer, we will have covered the code used in Chapter 1 but from within Pharo! We will add this method in the next chapter, when defining the class `LLVMMemoryBuffer`.

4.8 Summary

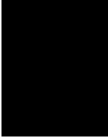
In this first part on Pharo bindings, we defined the basic bindings of the LLVM library. This is the beginning but some of the choices needed to be explained. Using a more object-oriented structure rather than a façade helps adding helper function or scale the binding library in the future, makes it easier to navigate through code and brings more familiarity from the Pharo environment within the LLVM world. For now, we can run the following piece of code and expect it to work:

```
| mod paramArray sumSig retType sum builder param1 param2 tmp |

mod := LLVMModule withName: 'mod'.
paramArray := LLVMParameterArray withSize: 2.
paramArray at: 1 put: (LLVMInt32 new handle getHandle).
paramArray at: 2 put: (LLVMInt32 new handle getHandle).
paramArray.

retType := LLVMInt32 new.
sumSig := LLVMFunctionSignature withReturnType: retType
    parametersVector: paramArray arity: 2 andIsVaridic: false.

sum := sumSig addToModule: mod withName: 'sum'.

builder := LLVMBuilder new.

param1 := sum parameterAtIndex: 1.
param2 := sum parameterAtIndex: 2.

tmp := builder buildAdd: param1 to: param2 andStoreUnder: 'tmp'.
builder buildReturnStatementFromValue: tmp.

>>> a LLVMValue((void*)@ 16r7F878F285658)
```

In the next chapter, we will focus on the memory buffer and the retranscription of the target and target machine. These are more complicated elements and they will need a bit more work in

order to operate properly.

CHAPTER 5



Advanced Pharo Bindings

5.1 Development plan

Following the previous chapter, we will continue our retranscription of the C code we managed to produce by the end of Chapter 2. This code allowed us in the end to obtain a file with the module we defined translated to machine code of a given architecture. In order to get to the same point from within Pharo, the last steps we need to do are the following:

- Memory buffer and emission
- Target creation
- Enumerations definition
- Target Machine creation

5.2 Memory Buffer and Emission

Following the same idea of what we did by the end of Chapter 2, we will need a way to output the module bitcode. This time, we do not need to direct it to a file but we would like to keep the representation inside the Pharo environment. We can do this by emitting the module bitcode to a memory buffer instead of a file. Therefore, let's first define the `LLVMMemoryBuffer`:

```
FFIExternalObject subclass: #LLVMMemoryBuffer
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Core'
```

Then, we can add the method transcripting the `LLVMWriteBitcodeToMemoryBuffer()` (LLVMWriteBitcodeToMemoryBufferhttps://llvm.org/doxygen/group__LLVMCBitWriter.html#ga43cccd6ab4fe5c042fc59d972430c97f*Then, we can add the method transcripting the 'LLVMWriteBitcodeToMemoryBuffer()' ([LLVMWriteBitcodeToMemoryBuffer](https://llvm.org/doxygen/group*LLVMCBitWriter.html#ga43cccd6ab4fe5c042fc59d972430c97f)). Adding this method will make more sense in the `LLVMModule` class and can be done as follows:

```
LLVMModule >> emitBitCodeToMemoryBuffer
  ^ self ffiCall: #(LLVMMemoryBuffer
    LLVMWriteBitcodeToMemoryBuffer(LLVMModule self))
```

We can now use the piece of code from the summary of Chapter 3 and add the emission to the memory buffer to get:

```
| mod paramArray sumSig retType sum builder param1 param2 tmp
    memBuff |

mod := LLVMModule withName: 'mod'.
paramArray := LLVMParameterArray withSize: 2.
paramArray at: 1 put: (LLVMInt32 new handle getHandle).
paramArray at: 2 put: (LLVMInt32 new handle getHandle).
paramArray.

retType := LLVMInt32 new.
sumSig := LLVMFunctionSignature withReturnType: retType
    parametersVector: paramArray arity: 2 andIsVaridic: false.

sum := sumSig addToModule: mod withName: 'sum'.

builder := LLVMBuilder new.

param1 := sum parameterAtIndex: 1.
param2 := sum parameterAtIndex: 2.

tmp := builder buildAdd: param1 to: param2 andStoreUnder: 'tmp'.
builder buildReturnStatementFromValue: tmp.

memBuff := mod emitBitCodeToMemoryBuffer.

>>> a LLVMMemoryBuffer((void*)@ 16r7F878F4A46B0)
```

This corresponds to what we managed to get by the end of Chapter 1, except it is now inside a memory buffer rather than written to a file.

5.3 Target creation

The `LLVMTarget` class will be tricky. If you remember correctly, we created the `Target` from a `Triple` string and we did that thanks to the function`LLVMGetTargetFromTriple()` (LLVMGet-TargetFromTriplehttps://llvm.org/doxygen/c_2TargetMachine_8h.html#a7a746a65818e0b6bd86e5f00a568e301*2TargetMachine*8h.html#a7a746a65818e0b6bd86e5f00a568e301)). What is new compared to the other classes? The function we would like to use does not return an `LLVMTarget` but gets as argument a pointer to an `LLVMTarget` that will be filled with the resulting target. Moreover, this function takes a string pointer (`char**`) to write the error if one appears.

In order to define pointers to objects from within Pharo, we have to define`FFIExternalValueHolders` as class variables. They can then be initialised from within an `initialize` class method:

```
FFIExternalObject subclass: #LLVMTarget
  instanceVariableNames: ''
  classVariableNames: 'LLVMTarget_PTR String_PTR'
  package: 'LLVMBindings-Target'
```

```
LLVMTarget class >> initialize
    LLVMTarget_PTR := FFIExternalValueHolder ofType: 'LLVMTarget'.
    String_PTR := FFIExternalValueHolder ofType: 'String'.
```

We can then call the function `LLVMGetTargetFromTriple` by using it as follows and calling our class variables:

```
LLVMTarget class >> getTargetIn: targetHolder fromTriple: triple
    errorMessage: errorHolder
    ^ self ffiCall: #(Boolean LLVMGetTargetFromTriple(const char*
    triple,
                                                  LLVMTarget_PTR
    targetHolder,
                                                  String_PTR
    errorHolder))
```

Then, a nice thing to do is to write a wrapper around the above method in order to get a coherent output in case of an error:

```
getTargetFromTriple: triple
    | targetHolder errorHolder hasError |

    targetHolder := LLVMTarget_PTR new.
    errorHolder := String_PTR new.

    "Returns 0 (false) on success"
    hasError := LLVMTarget getTargetIn: targetHolder fromTriple:
    triple errorMessage: errorHolder.
    hasError ifTrue: [ self error: errorHolder value ].

    ^ LLVMTarget fromHandle: targetHolder value.
```

There it is, we can define our target right? If you remember correctly from Chapter 2, one of the necessary step was to tell LLVM which targets we wanted to initialise before being able to actually use those targets. We got around this issue by using the functions `LLVMInitializeAllTargets`, `LLVMInitializeAllTargetInfos`,`LLVMInitializeAllAsmPrinters` and `LLVMInitializeAllTargetMCs`. However, those functions are defined as macros inside the C headers. This means that from C, they will be recognised as long as the header is included due to the fact that they will be processed by the C preprocessor. However, the only way to use them from Pharo would be to replicate their behaviour... Fortunately, what we will be able to do is to initialise the targets one by one. We will do this for one of them but can be replicated.

```
LLVMTarget class >> initializeX86Target
  self ffiCall: #(void LLVMInitializeX86Target())
```

```
LLVMTarget class >> initializeX86TargetInfo
  self ffiCall: #(void LLVMInitializeX86TargetInfo())
```

```
LLVMTarget class >> initializeX86TargetMC
  self ffiCall: #(void LLVMInitializeX86TargetMC())
```

```
LLVMTarget class >> initializeX86AsmPrinter
  self ffiCall: #(void LLVMInitializeX86AsmPrinter())
```

And finally, a gatherer of all the above methods:

```
LLVMTarget class >> initializeX86
  self initialize.
  self initializeX86Target.
  self initializeX86TargetInfo.
  self initializeX86TargetMC.
  self initializeX86AsmPrinter.
```

The exact same methods can defined for the architectures `AArch64` or `ARM` for example. We can now add some helper functions to interact with an `LLVMTarget` object:

```
LLVMTarget >> description
  ^ self ffiCall: #(const char *LLVMGetTargetDescription(LLVMTarget
    self))
```

```
LLVMTarget >> name
  ^ self ffiCall: #(const char *LLVMGetTargetName(LLVMTarget self))
```

5.4 Enumerations definition

Our target is set and ready. However, before being able to create a `TargetMachine`, we will need to define some enumerations. Enumerations are finite sets of data-types that can be used as a type later on. Here, when we will have to define a `TargetMachine`, we will have to choose an option from three different enumerations:

- `LLVMCodeGenOptLevel`: The level of optimisation we want, this is an enumeration we will have to choose from (`None`,`Less`,`Default` and `Agressive`)
- `LLVMRelocMode`: The mode of relocation we want, another enumeration we have to choose from (`Default`,`Static`,`PIC`,etc.)
- `LLVMCodeModel`: This is a term from AMD64 ABIhttps://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf (chapter 3.5) that defines the offset between data and code within instructions, also an enumeration we can choose from (`Default`,`JITDefault`,`Tiny`,`Small`,etc.)

We will go through the creation of the `LLVMCodeGenOptLevel` enumeration. First, let's define the class as a subclass of `FFIEnumeration`:

```
FFIEnumeration subclass: #LLVMCodeGenOptLevel
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'LLVMBindings-Enumeration'
```

Then, we will need to create a class method `enumDecl` that holds the different values:

```
LLVMCodeGenOptLevel class >> enumDecl
  ^ #(
  llvmCodeGenLevelNone       0
  llvmCodeGenLevelLess       1
  llvmCodeGenLevelDefault    2
  llvmCodeGenLevelAggressive 3
  )
```

Once this is done, run the `LLVMCodeGenOptLevel initialize` in order to properly create the enumeration. If everything went right, you should now have the following class definition:

```
FFIEnumeration subclass: #LLVMCodeGenOptLevel
  instanceVariableNames: ''
  classVariableNames: 'llvmCodeGenLevelAggressive
    llvmCodeGenLevelDefault llvmCodeGenLevelLess
    llvmCodeGenLevelNone'
  package: 'LLVMBindings-Enumeration'
```

The same thing can be done for the two other enumerations, here are the results:

- `LLCMRelocMode`

```
FFIEnumeration subclass: #LLVMRelocMode
  instanceVariableNames: ''
  classVariableNames: 'llvmRelocDefault llvmRelocDynamicNoPic
    llvmRelocPIC llvmRelocROPI llvmRelocROPI_RWPI llvmRelocRWPI
    llvmRelocStatic'
  package: 'LLVMBindings-Enumeration'
```

```
LLVMRelocMode class >> enumDecl
  ^ #(
  llvmRelocDefault      0
  llvmRelocStatic       1
  llvmRelocPIC          2
  llvmRelocDynamicNoPic 3
  llvmRelocROPI         4
  llvmRelocRWPI         5
  llvmRelocROPI_RWPI    6
  )
```

- `LLVMCodeModel`

```
FFIEnumeration subclass: #LLVMCodeModel
  instanceVariableNames: ''
  classVariableNames: 'llvmCodeModelDefault llvmCodeModelJITDefault
    llvmCodeModelKernel llvmCodeModelLarge llvmCodeModelMedium
    llvmCodeModelSmall llvmCodeModelTiny llvmodeModelDefault'
  package: 'LLVMBindings-Enumeration'
```

```
LLCMCodeModel class >> enumDecl
  ^ #(
  llvmodeModelDefault     0
  llvmCodeModelJITDefault 1
  llvmCodeModelTiny       2
  llvmCodeModelSmall      3
  llvmCodeModelKernel     4
  llvmCodeModelMedium     5
  llvmCodeModelLarge      6
  )
```

5.5 Target Machine Creation

We now have the enumerations we will need in order to properly create a`TargetMachine`. In order to make those enumerations available to the class, we need to add them as `poolDictionaries`. The resulting class definition is the following:

```
FFIExternalObject subclass: #LLVMTargetMachine
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: 'LLVMCodeGenOptLevel LLVMRelocMode LLVMCodeModel'
  package: 'LLVMBindings-Target'
```

The function `LLVMCreateTargetMachine()` (LLVMCreateTargetMachinehttps://llvm.org/doxygen/
c_2TargetMachine_8h.html#a9b0b2b1efd30fad999f2b2a7fdbf84922*TargetMachine*8h.html#a9b0b2b1efd30fad999f2b2a7fdb
can be written as follows:

```
LLVMTargetMachine class >> fromTarget: aTarget withTriple:
    aTripleString withCPU: aCPUString withFeatures: aFeaturesString
    withOptLevel: anOptimizationLevel withRelocMode: aRelocMode
    andCodeModel: aCodeModel
  ^ self ffiCall: #(LLVMTargetMachine
    LLVMCreateTargetMachine(LLVMTarget aTarget,
                                String
    aTripleString,
                                String              aCPUString,
                                String
    aFeaturesString,
                                LLVMCodeGenOptLevel
    anOptimizationLevel,
                                LLVMRelocMode       aRelocMode,
                                LLVMCodeModel       aCodeModel))
```

We will definitely not want to have to call this function with its seven different arguments. Let's add a default version wrapper:

```
LLVMTargetMachine class >> fromTarget: aTarget withTriple:
    aTripleString
  ^ LLVMTargetMachine fromTarget: aTarget withTriple: aTripleString
    withCPU: '' withFeatures: '' withOptLevel:
    llvmCodeGenLevelDefault withRelocMode: llvmRelocDefault
    andCodeModel: llvmCodeModelDefault
```

As you can see, the elements of the enumerations can be called simply by using them now that we have added the enumerations as used pool dictionaries. Going to the instance side, we will add some helpers as well as the classic `dispose` and`finalize` combo:

```
LLVMTargetMachine >> triple
  ^ self ffiCall: #(String
    LLVMGetTargetMachineTriple(LLVMTargetMachine self))
```

```
LLVMTargetMachine >> target
  ^ self ffiCall: #(LLVMTarget
    LLVMGetTargetMachineTarget(LLVMTargetMachine self))
```

```
LLVMTargetMachine >> cpu
  ^ self ffiCall: #(String LLVMGetTargetMachineCPU(LLVMTargetMachine
    self))
```

```
LLVMTargetMachine >> featureString
  ^ self ffiCall: #(String
    LLVMGetTargetMachineFeatureString(LLVMTargetMachine self))
```

```
LLVMTargetMachine >> dispose
  self ffiCall: #(void LLVMDisposeTargetMachine(LLVMTargetMachine
    self))
```

```
LLVMTargetMachine >> finalize
  self dispose
```

5.6 Memory Buffer Emission

The last step to cover the Pharo equivalent of Chapter 2 is to write the last binding for the emission to a memory buffer. First, we will need a last enumeration that will let us decide to what type of file we will like to output the code, having the choice between an object or assembly file.

```
FFIEnumeration subclass: #LLVMCodeGenFileType
  instanceVariableNames: ''
  classVariableNames: 'llvmAssemblyFile llvmObjectFile'
  package: 'LLVMBindings-Enumeration'
```

```
LLVMCodeGenFileType class >> enumDecl
  ^ #(
  llvmAssemblyFile 0
  llvmObjectFile   1
  )
```

Now, the final method we want to add is `LLVMTargetMachineEmitToMemoryBuffer()` (LLVM-TargetMachineEmitToMemoryBufferhttps://llvm.org/doxygen/c_2TargetMachine_8h.html#aaa9ce583969eb8754512e7
*getMachine*8h.html#aaa9ce583969eb8754512e70ec4b80061)). This function needs two pointers that we will have to implement the same way we did above with `getTargetFromTriple`. Let's rewrite the `LLVMModule` a bit:

```
FFIExternalObject subclass: #LLVMModule
  instanceVariableNames: ''
  classVariableNames: 'LLVMMemBuffer_PTR String_PTR'
  poolDictionaries: 'LLVMCodeGenFileType'
  package: 'LLVMBindings-Core'
```

We need to add the `LLVMCodeGenFileType` enumeration to the list of `poolDictionaries`. Moreover, you can see we defined two new class variables `LLVMMemBuff_PTR` and`String_PTR` (our error pointer).

```
LLVMModule class >> initialize
  String_PTR := FFIExternalValueHolder ofType: 'String'.
  LLVMMemBuffer_PTR := FFIExternalValueHolder ofType:
    'LLVMMemoryBuffer'.
```

We will have to call the `initialize` in our module creation method, `withName:`:

```
LLVMModule class >> withName: aName
  self initialize.
  ^ self ffiCall: #(LLVMModule LLVMModuleCreateWithName(String
    aName))
```

Now we can finally write the method to emit the module to a memory buffer given a proper target machine:

```
LLVMModule >> emitCodeFromTargetMachine: aTargetMachine
    toMemoryBuffer: aMemBufferPtr withFileType: aCodeFileType
    withError: anErrorHolder
  ^ self ffiCall: #(Boolean
    LLVMTargetMachineEmitToMemoryBuffer(LLVMTargetMachine
    aTargetMachine,
                                       LLVMModule self,
                                       LLVMCodeGenFileType
    aCodeFileType,
                                       String_PTR anErrorHolder,
                                       LLVMMemBuffer_PTR
    aMemBufferPtr))
```

Once again, a wrapper to correctly display the error if it is met is nice:

```
LLVMModule >> emitCodeFromTargetMachine: aTargetMachine
    withFileType: aCodeFileType

  | memBufferHolder errorHolder haveError |

    memBufferHolder := LLVMMemBuffer_PTR new.
    errorHolder := String_PTR new.

    "Returns 0 (false) on success"
    hasError := self emitCodeFromTargetMachine: aTargetMachine
    toMemoryBuffer: memBufferHolder withFileType: aCodeFileType
    withError: errorHolder.
    hasError ifTrue: [ self error: errorHolder value ].

    ^ LLVMMemoryBuffer fromHandle: memBufferHolder value.
```

Finally, we can write the last functions for each of the file type we know:

```
LLVMModule >> emitObjFromTargetMachine: aTargetMachine
  ^ self emitCodeFromTargetMachine: aTargetMachine withFileType:
    llvmObjectFile
```

```
LLVMModule >> emitASMFromTargetMachine: aTargetMachine
  ^ self emitCodeFromTargetMachine: aTargetMachine withFileType:
    llvmAssemblyFile
```

5.7 Summary

We covered in this chapter the last more advanced bindings in order to get the same behaviour as the one we were getting from C. Given our bindings, we should be able to make the following piece of code work:

```
| mod paramArray sumSig retType sum builder param1 param2 tmp
    memBuff  target targetMachine |

"MODULE DEFINITION"
mod := LLVMModule withName: 'mod'.

"PARAMETERS ARRAY DEFINITION"
paramArray := LLVMParameterArray withSize: 2.
paramArray at: 1 put: (LLVMInt32 new handle getHandle).
paramArray at: 2 put: (LLVMInt32 new handle getHandle).
paramArray.

"FUNCTION SIGNATURE DEFINITION"
retType := LLVMInt32 new.
sumSig := LLVMFunctionSignature withReturnType: retType
    parametersVector: paramArray arity: 2 andIsVaridic: false.

"FUNCTION DEFINITION"
sum := sumSig addToModule: mod withName: 'sum'.

"BUILDER DEFINITION AND OPERATIONS"
builder := LLVMBuilder new.

param1 := sum parameterAtIndex: 1.
param2 := sum parameterAtIndex: 2.

tmp := builder buildAdd: param1 to: param2 andStoreUnder: 'tmp'.
builder buildReturnStatementFromValue: tmp.

"TARGET DEFINITION"
LLVMTarget initializeX86.
target := LLVMTarget getTargetFromTriple: 'x86_64'.

"TARGET MACHINE DEFINITION"
targetMachine := LLVMTargetMachine fromTarget: target withTriple:
    'x86_64'.

memBuff := mod emitASMFromTargetMachine: targetMachine.
```