

The Meson Manual

Jussi Pakkanen

October 26, 2020

©2020 Jussi Pakkanen

The sample code in this book is provided for instructional purposes only. It has been tested with great care but it is not guaranteed to be free of errors or to work for any particular purpose.

Typeset by the author in L^AT_EX [11]. But given that you are the sort of person who reads colophons you probably knew that already.

3rd edition.

ISBN: 978-952-94-2892-2

Publisher: Diffraction Labs

Meson is a registered trademark of Jussi Pakkanen. All other trademarks are the property of their respective owners, and they are used only in an editorial fashion, with no intention of infringement.

Preface

When you spend a long time working on a book, deep, fundamental questions on the task start bubbling in to your mind. For this book some of these questions included *ggggnngh will this task never end, maybe I should vacuum my apartment one more time* and *what was the first ever build system in the world*. For the benefit of the reader we are going to ignore the first two and instead focus on the last one.

The obvious follow up question is *what is a build system*. If we go by modern nomenclature, then a build system is a tool whose job is to transfer source code from the user to the compilation process. This definition implies that the first build system was not software but instead a physical object, namely the cardboard box that operators used to carry punch cards from the keypunch station to the punch card reader. Thus we find that build systems are one of the oldest pieces of software development.

Given how fundamental a build system is for programming, it is quite astounding how little they have changed over the years. Let's take, for example, the time period between the years 1978 and 2000. Computers got thousands of times faster, many hardware companies, platforms and toolchains died, new one arose, flourished and then died again. Yet a software developer from the late 70s could feel quite at home, because most projects were built with Makefiles and shell scripts.

In the new millennium things started to change. Projects got bigger and people started requiring more things from their build systems, such as dependency management, unit testing and so on. Implementing all of this with Makefiles got harder and harder and eventually new solutions with wholly different approaches appeared. One of these was Meson, which was started just before Christmas 2012. There were two main reasons for its creation. The first one was that I was completely fed up with fixing the same things over and over again in other peoples' build definitions. The second was that I had not done a compiler course so I wanted to learn how to write a parser and interpreter for my own language.

That was the easy part.

The *really* hard part was convincing other people to use Meson. Here we finally discover that open source projects are not so much about the code, but about the people working on it. Over the years we have had hundreds, possibly even thousands of contributors ranging from code submitters to bug reporters, documentation writers and just plain old advocates. I would love to thank each and every one of you personally, but unfortunately there is not enough space here. I dedicate this book to all of you.

For their help in making this book, I'd like to thank the following people: Jukka Laurila, Juhani Simola, Tim Müller, Nirbheek Chauhan, Aleksandr Koltsoff. Their comments on earlier versions of this manuscript improved the end result immensely. However, as an old Finnish saying goes, one fool can create mistakes faster than ten smart people can fix, there are probably still some errors in the text. If you find any, please let me know via email so they can be fixed in the next edition.

Go forth and build!

Jussi Pakkanen
jpakkane@gmail.com
Espoo, Finland
January 2020

Preface to the 2nd edition

Thanks to Will Thompson, Tom-Robin Teschner and Will Wray for reporting errors in the first version of the book.

Preface to the 3rd edition

Thanks to Zlatko Karakas for reporting errors in the second edition.

Contents

Preface	iii
Conventions used in this book	xiii
I The user manual	1
1 Getting started	3
1.1 Obtaining Meson	3
1.2 Creating the sample project	5
1.3 Building from the command line	6
1.4 Building with the Visual Studio IDE	7
2 How compilation works	9
2.1 Basic term definitions	9
2.2 Building the Hello World application manually	11
2.3 Basic symbol resolution	12
2.4 Static linking	13
2.5 Shared linking	15
2.6 Linking multiple libraries	17
2.7 Which is better, shared or static linking?	19
2.8 Dynamic linker and symbol resolution	19
3 Meson syntax	23
3.1 Original design principles	24
3.2 Concrete design decisions	25
3.3 Elementary types	26
3.4 Build system phases	31
3.5 Program flow	32
3.6 Object types	41

3.7	Disablers	42
4	Building blocks of a software project	45
4.1	The elementary operations	46
4.2	Advanced build cases	47
4.3	Generating data	49
4.4	Defining the graph in Meson	49
4.5	Splitting the project to multiple directories	50
4.6	Target properties	52
5	External dependencies	55
5.1	What is a dependency?	55
5.2	Finding and using dependencies	56
5.3	Dependency provider backends	58
5.4	Executable dependencies	60
5.5	Dependencies that don't provide any dependency files	61
6	Subprojects and internal dependencies	63
6.1	Subproject basics and layout	63
6.2	Using subprojects	64
6.3	Internal dependencies	66
6.4	Combining subprojects and internal dependencies	66
6.5	Overriding executable lookup	67
7	Configuring the project	71
7.1	Simple approaches to configuration	71
7.2	Configuration files	74
7.3	Advanced configuration options	75
7.4	Introspecting the system	78
7.5	Printing status messages	81
8	Testing	85
8.1	Defining a test	85
8.2	Test properties	88
8.3	Advanced testing using the test tool	93
8.4	Defining custom test setups	96
8.5	Benchmarks	97
9	Installing	99
9.1	Directory layout	99
9.2	Installing build targets	102
9.3	Installing other files	103

9.4	Running the install	104
9.5	Custom install tasks	106
9.6	Other things that happen during install	107
9.7	Accessing data files before and after install	108
10	Project options	111
10.1	Builtin options	111
10.2	Declaring and using project options	112
10.3	Defining options	112
10.4	Exploring and setting option values	114
10.5	Sharing options between projects	116
11	Custom build steps	119
11.1	Generating data files	119
11.2	Dependency files	121
11.3	Special strings in command arguments	122
11.4	Generating source code	123
11.5	Generating source and headers	124
11.6	Using generators	127
12	Cross compilation	131
12.1	A word about nomenclature	132
12.2	A practical example	132
12.3	Other naming setups	135
12.4	Cross compilation with Meson	135
12.5	Cross file lookup	137
12.6	Multiple cross files	138
12.7	Constants in cross files	138
12.8	Native files	139
12.9	Running tests when cross compiling	139
12.10	Cross compilation and code generators	141
12.11	Firmware upload targets	142
13	The Wrap dependency download mechanism	145
13.1	The basic design	146
13.2	Downloading revision control checkouts	146
13.3	Downloading a release archive	147
13.4	Using the WrapDB	148

14	Converting an existing project to Meson	151
14.1	Why change build systems? Is it even worth it?	151
14.2	Making sense of an existing build system	153
14.3	Build tasks ordered by difficulty	155
14.4	Conversions involving an entire team	157
15	A library sample project	159
15.1	Design requirements	159
15.2	The external API	160
15.3	Precompiled headers	163
15.4	The C ↔ C++ bridge	164
15.5	Tests	165
15.6	Project layout	166
15.7	Creating releases	169
15.8	Exercises for the reader	171
16	Practical tips for real world projects	173
16.1	Use options rather than hardcoding compiler flags	173
16.2	Shipping pregenerated files	174
16.3	Do not treat files as strings	176
16.4	Running Python scripts that use extension modules	177
16.5	Move everything you can out of build files	179
II	The reference documentation	181
17	Elementary object reference	183
17.1	array	183
17.2	boolean	185
17.3	dictionary	186
17.4	disabler	187
17.5	integer	188
17.6	string	189
18	Domain specific object reference	195
18.1	build_machine	195
18.2	build_target	197
18.3	compiler	198
18.4	configuration data	207
18.5	custom_target	211
18.6	dependency	211
18.7	environment	214

18.8	external_library	215
18.9	external_program	215
18.10	generator	215
18.11	host_machine	216
18.12	meson	217
18.13	python_installation	222
18.14	run_result	224
18.15	source_configuration	225
18.16	source_set	225
18.17	target_machine	228
18.18	subproject	228
19	Function reference	229
19.1	add_global_arguments	229
19.2	add_global_link_arguments	230
19.3	add_languages	231
19.4	add_project_arguments	232
19.5	add_project_link_arguments	232
19.6	add_test_setup	232
19.7	alias_target	233
19.8	assert	234
19.9	benchmark	234
19.10	both_libraries	235
19.11	build_target	235
19.12	configuration_data	238
19.13	configure_file	238
19.14	custom_target	240
19.15	declare_dependency	242
19.16	dependency	244
19.17	disabler	245
19.18	environment	246
19.19	executable	246
19.20	error	247
19.21	find_program	247
19.22	files	248
19.23	generator	249
19.24	get_option	250
19.25	get_variable	250
19.26	import	251
19.27	include_directories	251
19.28	install_data	252
19.29	install_headers	253

19.30	install_man	254
19.31	install_subdir	255
19.32	is_disabler	256
19.33	is_variable	257
19.34	jar	257
19.35	join_paths	258
19.36	library	258
19.37	message	258
19.38	run_command	259
19.39	run_target	259
19.40	set_variable	260
19.41	shared_library	260
19.42	shared_module	261
19.43	static_library	262
19.44	subdir	262
19.45	subdir_done	263
19.46	subproject	263
19.47	summary	264
19.48	test	265
19.49	vcs_tag	267
20	Module reference	269
20.1	cmake	269
20.2	dlang	271
20.3	fs (filesystem)	271
20.4	gnome	274
20.5	hotdoc	281
20.6	i18n	282
20.7	pkgconfig	284
20.8	python	285
20.9	qt5	286
20.10	qt4	287
20.11	rpm	287
20.12	sourceset	288
20.13	windows	288
III	The appendixes	289
A	Contributing to Meson	291
A.1	Checking out the code	291
A.2	Creating the merge request	293

A.3	Review and continuous integration	294
A.4	Fixing merge conflicts	294
A.5	Documentation updates	295
Bibliography		297
Index		299

Conventions used in this book

This book has been designed to be as easy to read as possible. Unfortunately there is no universal definition for readability. What is simple for one person might be confusing for other people. In this chapter we describe the layout choices taken and how they should be read. We also list out some background requirements needed to get the most out of this book.

Programming languages

Meson supports many different programming languages. For simplicity this book focuses only on the C and C++ programming languages. The reader is assumed to have a basic understanding of their syntax and semantics.

Source code samples are typeset with a fixed width font. A small C++ program looks like this.

```
#include<iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, world\n";
    return 0;
}
```

Simplified function calls

Many function calls in Meson require several arguments. Sometimes this would produce a lot of noise that is not relevant to the issue being discussed. In these cases we use the following syntax:

```
some_function(...
    important_kwarg: 'value')
```

We use ... as a shorthand to hide all irrelevant arguments. The only thing that matters is the `important_kwarg` argument and its value.

Terminal output

At several points we need to examine the output of programs. Text written to a terminal is laid out like this:

```
$ ls -lah
drwxrwxr-x 2 jpakkane jpakkane 4,0K oct 17 12:49 getstarted
drwxrwxr-x 2 jpakkane jpakkane 4,0K oct 17 13:00 howtouse
drwxrwxr-x 2 jpakkane jpakkane 4,0K oct 17 12:48 installing
-rw-rw-r-- 1 jpakkane jpakkane 117 oct 17 12:54 book.aux
-rw-rw-r-- 1 jpakkane jpakkane 18K oct 17 12:54 book.log
-rw-rw-r-- 1 jpakkane jpakkane 89K oct 17 12:54 book.pdf
-rw-rw-r-- 1 jpakkane jpakkane 348 oct 17 13:00 book.tex
-rw-rw-r-- 1 jpakkane jpakkane 168 oct 17 12:54 book.toc
-rwxrwxr-x 1 jpakkane jpakkane 50 oct 17 11:28 compile.sh
-rw-rw-r-- 1 jpakkane jpakkane 18 oct 17 10:58 missfont.log
```

This sample demonstrates the two different parts of terminal text: commands written by the user and the output printed by programs. In the example above the first line is written by the user. These lines are printed in a *slanted font* and they begin with a dollar sign representing the command prompt (the \$ sign is not part of the command and should not be typed by the user). Program output is written in the a normal upright font.

Many Meson commands and output files contain file system paths. To improve readability these have been preprocessed so that path segments that point to a project's source and build directories have been replaced with `<<sourceroot>>` and `<<buildroot>>`, respectively. This syntax is used for hiding other such uninformative things, such as download urls and hash codes.

The output of commands has been preserved as much as possible. There are cases where this is not possible, usually due to output that is too wide to fit on a page. In these cases the text has been reformatted for a better visual appearance.

Operating systems

This book is mostly written from an *command line usage* perspective and assumes the reader to have basic fluency in using the Unix shell. Meson can be used directly from an IDE which hides all (or most) command invocations from the user. However since each IDE is slightly different it would be counterproductive to try to explain all of them in detail. Due to this we stick with the command line and point out IDE usage information only when necessary.

The samples also assume a Unix-like file system layout. For example the directory used for temporary files is `/tmp` and files are installed according to the Unix file system hierarchy to paths such as `/usr/bin` and `/usr/include`. People using operating systems with different file layouts need to mentally map these paths to the conventions of their chosen operating system.

Meson version

This book describes Meson version 0.55.0, which was released on the 12th of July, 2020.

Meson has had features in earlier releases that have been either deprecated or removed altogether. Those are not discussed as projects using those features should be updated to use new functionality. Documentation of these features can be found on the Meson project's web site for those who need them.

Part I:

The user manual

Chapter 1

Getting started

Compiling source code to programs can be quite difficult and a lot of work. But it does not need to be. Starting a project with Meson can be easy and even fun. We shall examine this by starting with a computer that has nothing but a compiler and going through all the steps needed to end up with a fully built program.

1.1 Obtaining Meson

The first step in using any piece of software is to obtain and install it. This can be done in many ways depending on your operating system and personal platform requirements.

1.1.1 System package manager

The simplest way of obtaining any software is to use the operating system's package manager, if one is available. This way has the least amount of hassle, and the system packagers have done the hard work of ensuring that it works with other packages provided by the system. For a build system this is especially important as it has to cooperate with many different low level programs. The installation requires typing only one command to a terminal.

```
$ sudo apt install meson
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  meson
```

4

```

0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 245 kB of archives.
After this operation,
  1 544 kB of additional disk space will be used.
Get:1 http://fi.archive.ubuntu.com/ubuntu cosmic/universe
      amd64 meson all 0.47.2-1ubuntu2 [245 kB]
Fetched 245 kB in 0s (1 644 kB/s)
Selecting previously unselected package meson.
(Reading database ... 372710 files and directories installed.)
Preparing to unpack ../meson_0.47.2-1ubuntu2_all.deb ...
Unpacking meson (0.47.2-1ubuntu2) ...
Setting up meson (0.47.2-1ubuntu2) ...
Processing triggers for man-db (2.8.4-2) ...

```

This example shows how to install Meson on an Ubuntu machine, but the same command works on all operating systems that derive from Debian. Other distros have different commands for installing system packages. See your distro's documentation for installation information.

1.1.2 Python's Pip tool

Sometimes system packages can not be used. Either because the current platform does not have a system package manager or because the version provided by the system is too old. In these cases a different approach is needed. As Meson is implemented in Python, it is also available through Python's Package Index. It can be installed using the `pip` tool, which is Python's own package manager.

```

$ pip3 install meson
Collecting meson
  Downloading <<download url>>/meson-0.50.0.tar.gz (1.4MB)
    100% |                               | 1.4MB 1.3MB/s
Building wheels for collected packages: meson
  Running setup.py bdist_wheel for meson ... done
  Stored in directory: <<pip dir>>
Successfully built meson
Installing collected packages: meson
Successfully installed meson-0.50.0

```

The command executed here is `pip3`, but it might have a different name on other platforms. Note that you should not use the command `pip` as it executes Pip for Python version 2 and Meson only supports Python version 3.

Pip might install its files to locations that are not in the system path, so they can not be run directly from the shell. A typical install location is `~/.local/bin/meson`. If this is the case for you, then you need to add the directory holding the Meson binary to the `PATH` environment variable before it can be used.

1.1.3 Windows binary packages

The Windows operating system does not ship Python by default. You can install it in the usual way and then install Meson via Pip as described in the previous chapter. This is not necessary, though, since the Meson project also provides fully standalone installer packages. These are in the Windows standard MSI format. Installing them is easy, after downloading the package, just double click on it to start the installer. The default options should result in a working installation.

1.2 Creating the sample project

First, let's run Meson with the `--help` command line switch to make sure that Meson has been installed properly and our environment is working.

```
$ meson --help
usage: meson [-h]
        setup,configure,install,introspect,init,test,wrap,help ...

optional arguments:
  -h, --help            show this help message and exit

Commands:
  If no command is specified it defaults to setup command.

  setup,configure,install,introspect,init,test,wrap,help
  setup                Configure the project
  configure            Change project options
  install              Install the project
  introspect           Introspect project
  init                 Create a new project
  test                 Run tests
  wrap                 Wrap tools
  help                 Print help of a subcommand
```

6

```
#include<iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, world!\n";
    return 0;
}
```

Figure 1.1: A C++ version of the well known Hello World application.

To get started we'll create a file called `hello.cpp`, which contains the standard hello world application as shown in Figure 1.1.

Then we create a *Meson build definition file* called `meson.build` with the following contents.

```
project('simple demo', 'cpp')
executable('hello', 'hello.cpp')
```

This is all we need to build our application, next we need to configure a build directory and then start the build.

1.3 Building from the command line

In the first step we invoke Meson and tell it what directory to use as the build directory. It will then set up everything needed to start the build.

```
$ meson build
The Meson build system
Version: 0.48.0
Source dir: /home/jpakkane/mesonbook/tmp/hello
Build dir: /home/jpakkane/mesonbook/tmp/hello/build
Build type: native build
Project name: simple demo
Project version: undefined
Native C++ compiler: c++ (gcc 7.3.0
    "c++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0")
Build machine cpu family: x86_64
Build machine cpu: x86_64
Build targets in project: 1
Found ninja-1.8.2 at /usr/bin/ninja
```

The project is compiled by invoking the Ninja [13] build tool.

```
$ ninja -C build
ninja: Entering directory `build'
[2/2] Linking target hello.
```

7

We used the `-C` command line switch to Ninja to tell it where the build directory is. An alternative approach is to first go in the build directory with `cd build` and then invoking Ninja without any command line arguments.

Now the project has been built and we can run the resulting binary directly.

```
$ build/hello
Hello, world!
```

1.4 Building with the Visual Studio IDE

Visual Studio [14] behaves in a different way than most compiler toolchains, especially ones on Unix type platforms. Its compilers are not in path by default so you can't use them directly. The simplest way is to launch the *Developer command prompt* that can be found in the Start menu under Visual Studio's application folder.

Once the terminal has been launched, Meson can be run in a similar way as above.

```
$ meson build --backend=vs
The Meson build system
Version: 0.48.999
Source dir: C:\Users\IEUser\hello
Build dir: C:\Users\IEUser\hello\build
Build type: native build
Auto detected Visual Studio backend: vs2017
Project name: hello
Project version: undefined
Native C++ compiler: cl (msvc 19.15.26730)
Build machine cpu family: x86_64
Build machine cpu: x86_64
Build targets in project: 1
```

The only difference is the `--backend=vs` command line argument. We use it to tell Meson that we want to generate a Visual Studio Solution. Meson defaults to using Ninja, and latest versions of Visual Studio provide a copy if you install

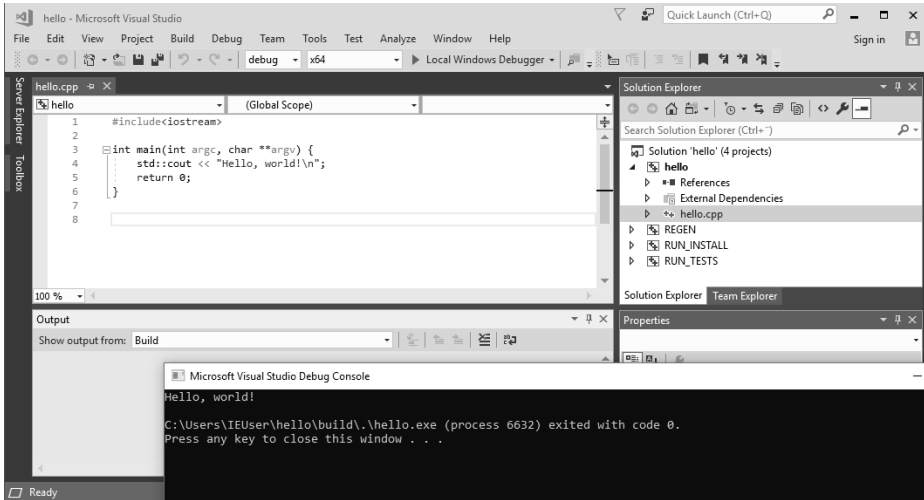


Figure 1.2: Running the built hello application from the Visual Studio IDE.

certain extensions. Thus we need to have this argument to ensure that IDE project files are generated.

Meson will generate a *Visual Studio solution file* in the build directory, which is the native project format for this IDE. Figure 1.2 shows the result of opening the solution in the IDE, building it and running the resulting executable. The debug console window is opened automatically by the IDE for all console applications so their output can be seen.

Chapter 2

How compilation works

Compiling source code into executables looks fairly simple on the surface but gets more and more complicated the lower down the stack you go. It is a testament to the design and hard work of toolchain developers that most developers don't need to worry about those issues during day to day coding.

There are (at least) two reasons for learning how the system works behind the scenes. The first one is that learning new things is fun and interesting *an sich*. The second one is that having a grasp of the underlying system and its mechanics makes it easier to debug the issues that inevitably crop up as your projects get larger and more complex.

This chapter aims to outline how the compilation process works starting from a single source file and ending with running the resulting executable. The information in this chapter is not necessary to be able to use Meson. Beginners may skip it if they so choose, but they are advised to come back and read it once they have more experience with the software build process.

The treatise in this book is written from the perspective of a build system. Details of the process that are not relevant for this use have been simplified or omitted. Entire books could (and have been) written about subcomponents of the build process. Readers interested in going deeper are advised to look up more detailed reference works such as chapters 41 and 42 of [10].

2.1 Basic term definitions

compile time All operations that are done before the final executable or library is generated are said to happen during *compile time*. Some people use the term informally and include linking in compile time. Others are more strict and use the term *link time* to distinguish between the two.

run time All operations that happen once a built executable is run are said to happen during *run time* (sometimes also called *runtime*). In this chapter we are mostly interested in the run time behaviour of symbol resolution via dynamic linking.

source file Source files contain the actual source code that programs are made of. They usually have file name extensions such as `.c`, `.java` or `.cpp`.

header file Some languages have separate header files that contain things such as function and variable declarations (but not their definitions). In C++ code that uses templates this gets a bit murkier. It is possible, and in fact quite common, to have code inside header files, but for the purposes of this chapter we can mostly ignore it.

object file An object file is the intermediate step between a source file and an executable or library. The compiler converts one source file into one object file, which contains machine executable binary code. An object file is not usable on its own until it is linked to a build target.

compiler A compiler's job is to take source files, parse their contents and generate corresponding binary code. It is also responsible for optimising the output, printing warnings, generating debug information and sometimes even doing static analysis on the source code.

linker The task of taking built object files and dependency libraries and assembling that into a cohesive whole, usually either a shared library or an executable, falls to the linker. Few people need to deal with the linker directly, and on most platforms it is invoked via the compiler.

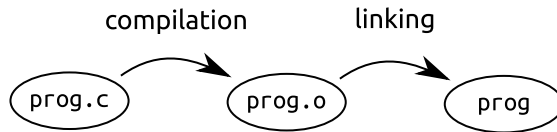
static linker A static linker is a tool that produces static libraries from object files.

symbol Many things in binary code have names by which they can be identified. Examples include functions and global variables. These names are called *symbols*.

static library A static library is an archive file containing only object files.

shared library A shared library is a fully built piece of code that other programs can use. When a program (or library) is linked against a shared library no code is copied. Instead the linker stores the name of the dependency library in the target it is building to be used later by the dynamic linker at runtime.

executable An executable is a program that can be run.



11

Figure 2.1: The compilation steps needed for a program consisting of one source file.

dynamic linker An executable using shared libraries can not be executed directly. Some process must find the libraries it needs and map all symbols used by the executable to their respective locations. This task is handled by the dynamic linker. It does not come from the compiler toolchain, but is a core component provided by the operating system.

2.2 Building the Hello World application manually

A simple way to get started is to compile a simple program manually. For this we'll use the helloworld application presented in Figure 1.1. As discussed above, the build process can be split into two separate parts: the *compilation step* and the *linking step*. The workflow is visualised in Figure 2.1. The compilation is done by invoking the compiler.

```
$ c++ -c -o hello.o hello.cpp
```

By default most compilers want to compile and link the entire application in one step. We have to use the `-c` command line argument since we only want to compile. The output goes to the object file `hello.o`.

Linking is just as simple.

```
$ c++ -o hello hello.o
```

We don't call the linker binary (which on this platform is called `ld`) but instead use the compiler to do the linking for us. The reason for this becomes fairly obvious if we look at the command line needed to link the program with plain `ld`.

```
ld -o hello \  
-dynamic-linker /lib64/ld-linux-x86-64.so.2 \  
/usr/lib/x86_64-linux-gnu/Scrt1.o \  
/usr/lib/x86_64-linux-gnu/crti.o \  
/usr/lib/x86_64-linux-gnu/crtn.o
```

```
/usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o \  
hello.o \  
-L/usr/lib/x86_64-linux-gnu \  
-L/usr/lib/gcc/x86_64-linux-gnu/7/ \  
-lstdc++ -lgcc -lc -lgcc_s \  
/usr/lib/gcc/x86_64-linux-gnu/7/crtendS.o \  
/usr/lib/x86_64-linux-gnu/crtn.o
```

These command line arguments specify all sorts of functionality needed to talk with the core operating system and by the C++ language runtime. Unless you are working on the compiler toolchain or other such low level component, it is unlikely you'll ever need to deal with the linker manually.

The built executable can now be run.

```
$ ./hello  
Hello, world.
```

2.3 Basic symbol resolution

From the developer point of view, compilation is fairly straightforward and easy to comprehend. Source code goes in and a binary artefact comes out. Linking, on the other hand, is a lot more vague. The most user visible operation that happens during linking is *symbol resolution*. In order to understand it, we must first examine what symbols are in the compilation context.

Symbol resolution happens at a very low level, and thus it is necessary to go all the way down to assembly code to understand its behaviour. A simple source file and its corresponding assembly output can be seen in Figure 2.2. Understanding exactly what the individual assembly instructions do is not necessary, a rough understanding of the overall structure is sufficient.

A symbol is nothing more than a string which specifies a name of a thing in the program. To keep things from getting too simple, not all symbols have a name and some names do not have a corresponding symbol. The sample code has three different named elements: the `print_number` and `printf` functions and the `number` variable. The first two of these have a symbol name but the variable name does not. This is because the linker only works with elements that are in global scope, that is, functions and global variables and constants. Both of these names can be found in the assembly output.

The element that does have a symbol but not a name is the character array `"Number %d\n."`. This may seem surprising given that the character array is only used inside the function just like the `number` variable. What happens behind the scenes is that the compiler elevates the character array to a global constant

<pre>#include<stdio.h> void print_number(int number) { printf("Number %d.\n", number); }</pre>	<pre>.LC0: .string "Number %d.\n" print_number: mov esi, edi xor eax, eax mov edi, OFFSET FLAT:.LC0 jmp printf</pre>
---	--

13

Figure 2.2: A simple C function (left) and the result of compiling it to x86_64 assembly (right).

and gives it a secret symbol name, which in this case is `.LC0`, as can be seen at the beginning of the assembly output. Effectively it is as if the compiler had compiled a program that looks like this:

```
#include<stdio.h>

/* Leading dot removed, because variable names can not
 * have the character "." in them.
 */
const char LC0[] = "Number %d.\n";

void print_number(int number) {
    printf(LC0, number);
}
```

At this point the compiler's job is finished and the object files are handed to the linker. Its job can be most easily understood by looking at the last line of the assembly code which is `jmp printf`. This is the call to the `printf` function which is part of the standard library. Sadly processors do not understand textual labels, they can only jump to specific memory addresses. The main task of the linker is to go through the compiled code and replace all references to symbols with numerical addresses that point to the corresponding functions and global variables. If all symbols used by the program are found the program can be generated and run. In case any piece of code tries to use a symbol that the linker can not find, the linker will abort with an error.

2.4 Static linking

Thus far we have only looked at single executables where all source code is compiled and linked directly. In real world projects this setup is fairly rare. Most

applications use code that has been built separately. A collection of prebuilt code is called a *library*. There are two different kinds of libraries, *static libraries* and *shared libraries* and using code from these on a target is called *static linking* and *shared linking*, respectively. We shall first look at static linking, since it is the simpler of the two.

To demonstrate linking we are going to need two things: a library and an executable using it. We'll create our own library called *messageprinter*. It consists of one file, `messageprinter.c`.

```
#include<stdio.h>

void print_message() {
    printf("I am a library.\n");
}
```

The only thing this function does is print a message to the screen proving that the function has been called. A main program using the library is equally plain.

```
void print_message();

int main(int argc, char **argv) {
    print_message();
    return 0;
}
```

The only thing to note is that `main.c` manually specifies the function prototype at the beginning rather than by including a header. This is merely to simplify the code.

This is all that we need to build and run an executable using static linking. Building the static library takes two commands.

```
$ cc -c -o messageprinter.o messageprinter.c
$ ar csrD libmessageprinter.a messageprinter.o
```

The first command is the familiar compiler invocation. The second command is where the library gets built. It is done with the `ar` command, which is known as the *static linker*. This is actually a misnomer, since `ar` does not do any linking at all. The only thing it does is take all the specified object files and put them together in an archive file. Its behaviour is almost identical to other archive program such as `tar` and `zip`. Because of this the static linker is sometimes called a *static archiver*. The library file name is `libmessageprinter.a`. The

standard way of naming libraries is to have the `lib` prefix and `.a` as the file extension. This is not mandatory, the archive can have any name, but most tools, processes and developers expect this naming scheme so you should use it unless there are strong reasons for doing something else.

The library can be used by adding it on the final executable's link command line.

```
$ cc -c -o main.o main.c
$ cc -o main main.o libmessageprinter.a
$ ./main
I am a library.
```

In addition to passing the library directly, there is an alternative syntax that is used especially for libraries provided by the system.

```
$ cc -o main main.o -L. -lmessageprinter
```

This way of using the library requires two command line arguments. The latter one is `-lmessageprinter` which tells the linker to *find a library called messageprinter, following the standard naming scheme, and link against that*. The standard naming scheme is the one mentioned above. If the library file is not `libmessageprinter.a`, the linker could not find it and linking would fail. By default the linker only does lookups in the *system library directories*. Since our library is not in one of those, we need to add the current directory to the list of lookup directories with the `-L.` command line argument.

The algorithm the linker uses to handle static libraries is straightforward. If it finds that some object file contains a symbol needed by the main program, it will copy out that object file and link all of it with the main program. The behaviour is the same *as if* you had manually specified those object files to be linked like this:

```
$ cc -o main main.o messageprinter.o
```

In this simple case the entire contents of the static library is used. But if the library contains many object files, only the ones whose symbols are actually needed (and their transitive dependencies) end up in the final executable. If only a small fraction of the library's code is needed, this can lead to noticeable space savings in the final executable.

2.5 Shared linking

Building and using a shared library is not very different from static linking.

```
$ cc -o messageprinter.o -fPIC -c messageprinter.c
$ cc -o libmessageprinter.so -shared messageprinter.o
```

The only difference to static linking are the output filename and the two command line arguments. The compiler argument `-fPIC` tells the compiler that the object file will be used in a shared library so it must be built as *position-independent code*. What this means will be explained later in this chapter. On many platforms this argument is not needed as all code is built position-independent by default but we use it here for portability. The linker argument `-shared` tells the linker that the output file should be a shared library.

Linking the main program with the shared library is almost identical to using a static library.

```
$ cc -o main main.o libmessageprinter.so
```

You can also use the `-L. -lmessageprinter` syntax, which works in the same way. If you try to run the result, you will get a mysterious crash:

```
$ ./main
./main: error while loading shared libraries:
libmessageprinter.so:
cannot open shared object file: No such file or directory
```

This error stems from the main difference between static and shared libraries. Shared libraries are full featured operating system components whereas static libraries are only archives of objects. The former can be used in various ways during runtime but the only thing you can meaningfully do to a static library is link it to an executable or a shared library.

In static linking all object code used by the application is copied to the target executable. In shared linking this does not happen. Instead the shared library's name is written in the executable's *dynamic section*. It contains a list of all external shared libraries required to run the program. No code from the shared library is copied inside the executable. When the program is run it is the responsibility of the operating system's *dynamic linker* to find all shared libraries needed by the program and to resolve all missing symbols. This lookup is done every time the program is run.

Just like the static and shared linkers need to be told where to look up libraries, the dynamic linker needs to be told where to look up shared libraries. Due to security reasons the current directory is not in the library search path by default¹. We need to add it to the list with the `LD_LIBRARY_PATH` environment variable.

¹Just like the current directory is not in `PATH` by default.


```
$ LD_LIBRARY_PATH=. ./main
I am a library.
```

17

One unexplained question about this program remains about the program's use of `printf`. Since it is a symbol and all symbols need to be resolved before a program can be run (both when shared and static linking), where does that symbol come from? To find this out we need to look inside the produced executable. There are many tools available to inspect the contents of programs. We'll use the `ldd` program that lists all libraries needed by an executable.

```
$ LD_LIBRARY_PATH=. ldd main
linux-vdso.so.1 (0x7ffd3e9f7000)
libmessageprinter.so => ./libmessageprinter.so (0x7f741e7f1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x7f741e400000)
/lib64/ld-linux-x86-64.so.2 (0x7f741ebf5000)
```

Even though we specified one shared library, the final executable ended up with four of them. `linux-vdso.so.1` is a performance optimisation mechanism that makes certain Linux system calls faster. The second line contains the shared library we just built.

The third one is `libc.so.6`. This is where `printf` actually comes from. This library is called the *C runtime library* and contains all functionality needed by C programs, such as `malloc` for allocating memory. It also contains code needed for process startup and teardown. Most programming languages have a similar runtime library that they link dynamically against their programs. There are also languages that don't behave in this manner.

The final entry is `/lib64/ld-linux-x86-64.so`. This is the system's dynamic linker. It may seem bizarre that the dynamic linker, whose job is to find an executable's shared dependency libraries is itself provided as a shared library. The answer to this chicken and egg problem is that the dynamic linker is not a dependency library in the traditional sense, `ldd` merely reports it that way. In reality it is set up as the program's *ELF interpreter*. A detailed description of the issue is out of scope for this book, but interested readers can find more information in the ELF reference documentation [18].

2.6 Linking multiple libraries

Linking multiple libraries is only slightly more complex than only one. Basically the linker will proceed through the items on the link line one by one until all linker targets have been processed. Thus far the behaviour of all operating

systems and toolchains has been almost identical, but this is where they start to differ noticeably.

For the purposes of this discussion, let's assume that we have a project that consists of one main object file and two libraries called **one** and **two**. The main object uses functionality from library **one** which in turn uses functionality from library **two**.

2.6.1 The classical Unix linking model

The sample project would eventually be linked using the following command:

```
$ cc -o main main.o libone.a libtwo.a
```

Here we use static libraries. The way the linker goes about its job is that it starts by taking the first argument **main.o**. It processes the file and makes a list of all external symbols that it requires. Then it processes the next argument **libone.a**. For each symbol in the missing symbol list it will try to see if any of the object files inside the library provides it. If yes, it will copy the object file out as described in section 2.4.

Once the linker has satisfied as many missing symbols as it possibly can, the static library is discarded. Any object files that were not needed to satisfy symbols are thrown away. After that the linker will go to the next argument **libtwo.a** and repeat the process. If all symbol requirements were found, the linking step is a success, otherwise an error is raised.

The main problem with this algorithm is that it is fragile and sometimes confusing. It breaks if you get the order of libraries wrong. This will not link:

```
$ cc -o main main.o libtwo.a libone.a
```

The reason is simple. When **libtwo.a** is being processed, none of the symbols it provides are in the list of needed symbols. That means that *everything* in it gets thrown away. When **libone.a** is processed those symbols are added to the list, but they can't be fulfilled any more because **libtwo.a** is gone.

This was a fairly common problem back when Makefiles were written by hand. It is very confusing to be told by the linker that your program has unresolved symbols even though you can clearly see them on the linker command line. This led to lots of “cargo cult” problem solving where developers would add the same libraries on the command line many times in the hopes that eventually it would work.

Sometimes it is even necessary to have the same dependency library on the command line multiple times. This happens if you have a circular dependency between two libraries. This happens when library A requires symbols from

library B and vice versa. If this ever happens the only reasonable approach is to change the code so the circular dependency is broken. If this is not possible for some reason, then the only workable solution is to tell the linker to first link A, then B and then A again. Or possibly B, A, B depending on how the calling program uses the libraries. In fact for pathological cases there may be an arbitrary number of repetitions needed. It is left as an exercise to the reader to work out how that might come about.

The reason for this behaviour is that linkers were originally designed and implemented in the early 70s. At the time computers were slow and had little memory. Keeping all symbols alive would have required too many resources, and actively reducing the amount of data to keep in memory made sense. Then, as it usually happens, computers got a lot faster so this limitation was no longer an issue, but the behaviour was kept to maintain backwards compatibility.

The most widespread linker in current use that behaves like this is the GNU bfd linker, which is the default on most Linux distributions.

2.6.2 Modern linker model

Modern linkers behave in roughly the same way as the classical Unix linker, except that they don't discard any libraries. This means that symbol resolution happens globally. It does not matter which order the libraries are defined, because the linker will search for symbols in every file.

The exact order in which symbols are resolved depends on each linker. Usually developers do not have to care about it as long as no symbol is repeated. Duplicated symbols are considered an error. Most new linkers behave in this manner, including the Visual Studio linker, macOS linker and the lld linker provided by the LLVM project.

2.7 Which is better, shared or static linking?

This is a common topic of, shall we say, lively debate on the Internet. Both of these approaches have their merits and use cases. Meson does not have a preference, instead it aims to work identically with both library types. Switching between the two library types is simple, as the only change needed is to alter the library target's type.

2.8 Dynamic linker and symbol resolution

Now that we know the difference between static and shared linking we can examine how symbol resolution works in more detail. We will remain at the

conceptual level, though. Readers interested in the actual implementation details are instructed to look up more detailed reference works such as [5].

Resolving symbols in static linking is not particularly complicated. As was discussed in Section 2.3 the compiler will write placeholder code for all symbols outside the current translation unit. In addition it writes a set of *relocation records*. They are merely a list of locations of said placeholders and which symbol's address they should be filled in with. When the final executable is linked, the linker has all the code and thus the addresses of all symbols. It can then overwrite the placeholders with the real addresses.

Dynamic linking is more difficult. Nothing about it is known at link time apart from its name. We don't know what address it will end up when the program is run due to *address layout randomisation* or ASLR. This is a security mechanism against various memory corruption vulnerabilities. Whenever a piece of code is loaded into memory, whether it comes from an executable or a shared library, it is placed at a random address in the process' virtual address space. Thus some sort of an indirection mechanism is needed to make things work.

Suppose we build an executable called `proggy` in the current directory and that it uses a shared library `thingy` which resides in directory `subdir`. The program would be built with the following command line invocation:

```
$ gcc -g -o proggy main.c subdir/libthingy.so
```

The linker adds an entry to the executable that it requires `libthingy.so` to run. This can be verified with the `ldd` command.

```
$ ldd proggy
...
subdir/libthingy.so (0x00007f5bfe106000)
```

As can be seen, the entry also contains the subdirectory where the library resides in. To keep things from being too simple and straightforward, this depends on the compiler flags used. If the executable is linked with the alternative link syntax like this:

```
$ gcc -g -o proggyL main.c -Lsubdir -lthingy
```

then only the filename is written in the executable:

```
$ ldd proggy
..
libthingy.so => not found
```

This behaviour is confusing and is probably inherited from the 70s and can not be changed due to backwards compatibility. The solution to this is an entry called *soname*, which is a “virtual file name” that can be defined for each shared library. There are exact rules on how sonames should be determined but Meson will do that automatically.

When an executable that uses shared libraries is run, it is the responsibility of the dynamic linker, sometimes also called a *loader*, which starts with the main executable and the list of sonames that it requires. It will search for shared libraries matching the sonames on the system in a platform specific way. Any libraries required by the found shared libraries are also looked up in the same way. If any of the libraries can not be found, then the process is not run, but instead exits with an error.

Now the dynamic linker is almost at the same position as we were when linking the executable statically. It has all the symbol names and knows the corresponding runtime addresses. It could, in theory, write the actual addresses in the code that has been loaded in memory, but it turns out that this can’t be done. All code loaded from files is mapped to memory as *read-only* so the actual code can’t be changed. This is due to performance and security reasons. Thus an additional piece of functionality is needed.

For function calls this is done with a data structure called the *procedure linkage table* or *PLT*.² A slightly simplified way of looking at it is to consider it as a table of function pointers, one for each symbol needed. Once the table is filled, the code can call any function it needs to execute. Yet, the tables are not filled yet.

The ELF file format used by most unixes is very powerful and flexible and supports many different ways of loading, using and interposing symbols. We shall not look at them in detail, but what is important for this discussion is that symbol loading is done *lazily*. That is, the actual location of any symbol is only determined when someone actually calls it. This also improves program startup times, since it is not uncommon for programs to only use a subset of all symbols at runtime. Symbol lookup takes time, and resolving all symbols up front would be slow. An outline of the lookup process can be seen in Figure 2.3.

Symbol resolution starts by the executable calling a function that resides in some shared library. This is implemented by calling the function pointer in the PLT that corresponds to the desired function. This reduces to calling a function pointer in the PLT (which is an array) with an offset. All of this information was available when the executable was built, so this can be done directly.

As discussed earlier, the PLT does not contain function pointers to the actual code. Instead all pointers in the PLT have been set to point to a symbol

²Global variables are looked up in an analogous fashion using a table called *global offset table* or GOT.

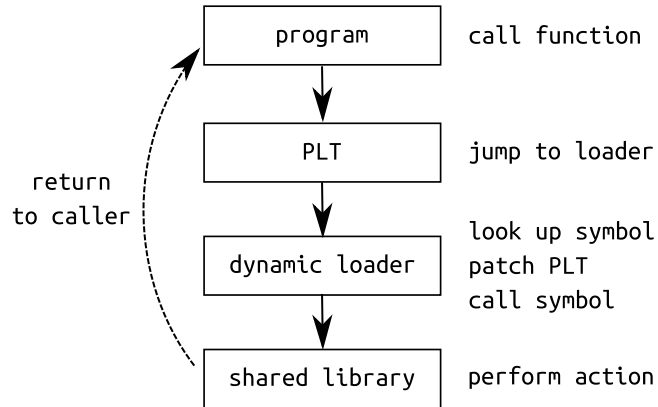


Figure 2.3: How the dynamic linker looks up symbols.

resolution function inside the dynamic loader and to pass it an argument specifying which function in the table initiated the call. The dynamic loader now knows which function was called and can look up its actual runtime address. It writes this address to the PLT entry and then jumps to that function. From the outside it looks as if the program had called the correct function in the shared library directly.

When the function is called a second time, the entry in the PLT already points to the correct location. The expensive resolution operation is thus done only once and only for those symbols that have actually been used.

Chapter 3

Meson syntax

The previous chapters should provide a rough understanding of using Meson and what is required of a build system. Armed with this information we can now dive into the core of Meson: its build definition language.

Traditionally build system definition languages have been fairly low level. It was common to have to type compiler flags, command lines and the like by hand. A significant fraction of build systems still in use today provide only a single data type: a string of characters. Meson works at a higher level of abstraction to simplify the task of writing build definitions but without losing expressive power. This is achieved by providing a richer set of general and domain specific types.

Many programming languages such as C++ are defined in terms of an *abstract machine*. Rather than try to add every computer architecture's idiosyncrasies in the definition, the language is instead defined in terms of an idealised computer. It is the task of the compiler to take a program written for this hypothetical computer and convert it to run on a real piece of hardware. The output (and side effects desired by the author) of the actual program must be the same as for the idealised version. This makes the programmer's job a lot simpler, especially for projects that need to work on multiple platforms. The downside is that this makes writing the compiler a harder task. The same principles apply for Meson.

The end user's task is to write a *build definition* of their project using the high level constructs provided by the build system. These include libraries, executables, external dependencies, source files and other types. Meson will then take this definition, inspect it, validate it and eventually convert it to individual elementary actions such as compiler and linker invocations.

Most of the time this high level of abstraction is sufficient. There are many cases, though, where it is necessary to dive deeper and deal with the underlying details, such as specifying specific compiler flags. Meson aims to provide a good user experience for both these cases. In fact this duality is the root cause of

most of Meson's internal implementation complexity. The payoff for this work is that from the end user point of view Meson stays close to the golden rule of design:

Simple things should be simple.

Hard things should be possible.

3.1 Original design principles

Build systems have not traditionally been known for their nice and readable syntax. Meson was originally designed from the ground up with usability in mind. Build systems are quite different from other development tools such as editors and debuggers, in that when you are interacting with the build system, you are *not productive*. That is, every moment spent dealing with build definitions is, strictly speaking, time lost. A driving force in build system design is to *minimize* the time users are dealing with it.

The first, and perhaps the most important, design principle is that build definitions should be readable. More specifically, when a user reads just one build definition file in a large project, they should obtain a fairly good understanding of what the definition does. There should not be “spooky action at a distance” where setting a variable in a random other build definition file unexpectedly alters the behaviour of seemingly unrelated build steps.

Many build systems were born in or have a legacy from the 70s. At the time terseness and use of “magic” characters was common and considered good practice. The downside is that using them means memorising what cryptic character combinations such as `$<` and `?:` mean, and how `$*` differs from `%`¹. Since the list of special characters is limited, most Unix programs repurpose them to their own use. Many character combinations behave slightly differently in different tools, so the end user has to remember what a given character combination does in the *specific* program they are using. Because of these problems a general trend for designing text based formats has shifted to use plain English keywords instead. This makes code easier to read at the cost of taking more keystrokes to write.

Using plain words in build definitions is not without its problems, either. The obvious pothole is to make things too verbose. This makes reading and understanding build definitions obtuse and tiring. Meson aims to strike a balance between these two extremes by using plain text keywords as much as possible while using special characters sparingly. In practice this mostly means using

¹For plain Make these are, *inputs to current target*, *assign value if not already existing* and *\$* is the full name of the target whereas % is the “matched part” of the target filename*, respectively.

established computer science practices, such as using `[]` for arrays and indexing, and `{}` for dictionaries.

One final point is that build definition should not look like COBOL, if at all possible. THE RATIONALE FOR THIS DESIGN CHOICE SHOULD BE FAIRLY EASY TO SEE AT A GLANCE.

3.2 Concrete design decisions

Coming up with solid design principles is difficult, but that is only one half of the job. The second part is creating the actual product while keeping true to the original vision. This is also the trickier part of the two, because implementing things in the real world turns up lots of edge cases and complications that were not apparent during the original design.

The most important design choice (and in fact the first) is that the build definition language must not be *Turing-complete* [19]. What this roughly means is that the Meson definition language is not really a programming language as such. It can not express an arbitrary program. In practice this means that you can not define functions, macros or arbitrary loop constructs.

Meson is at its core an *imperative* definition language. That is, it proceeds along the build definition one line at a time in sequence. This is in direct contrast with many other build systems, most famously Make, that are fully *declarative*. In these systems you don't describe a series of steps to execute but instead merely a set of constraints, which the build system will then evaluate in any way it seems fit. Purely declarative systems are two edged swords. On one hand when they work they can be extremely terse, readable and performant. On the other hand when there are bugs in the declarations the debugging experience may be exhausting. Fortunately declarativity is not an *either/or* proposition and it is possible to use declarative elements in iterative systems. In the spirit of Albert Einstein it could be said that Meson aims to be *As declarative as possible but no declarativer*.

Another major design choice is that all created objects are *immutable*. They can not be altered in any way (even deleted) once created. It follows that all information needed to create an object must be present when it is created. This makes it easy to see what goes into any object, which again simplifies understanding build definitions. There are a couple of object types that can be modified after creation, these exceptions to the rules will be discussed later.

An unfortunate truth about build systems is that the real world of software development can get quite messy. Examining build definitions of large scale existing projects turns up a lot of custom functionality that has been put inside build definition files. These include things such as shell pipelines for processing files and querying information about the current system. Having the code inside

build files makes them hard to read, not to mention that all special characters must be quoted to make them work, including the quoting of quote characters.

In Meson having this kind of code inside the build definition file is considered an antipattern. Instead all such functionality should be put in standalone script files that can then be invoked from `meson.build` files. This is the “escape hatch” that makes it possible to run arbitrary code while keeping the main build definition clean and understandable. It also means that the external scripts can be written in any programming language: Unix shell, Python, Windows `.bat` files or anything else. From the build system point of view they are indistinguishable.

These design choices all make sense on their own. But the true power of them comes when they are put together. The most illuminating example is the combination of immutability and the requirement that all information about a target must be presented at the point of object construction. What this means is that *circular dependencies are impossible to express*.

In traditional and fully declarative systems you have to refer to things (dependencies in this case) with a reference, which is typically a string holding the dependency’s name. It is easy to accidentally create a dependency loop. To define a dependency between two objects in Meson you must provide the actual object itself as an argument. There is no way to create an object that refers to a different object that will come to existence at some later point in time and since objects are immutable such a loop can not be created after the fact either. Thus it is impossible to express a loop between two objects. Declarative systems have a cycle detector that errors out when the user has written a cyclic dependency, but build definitions in Meson are guaranteed to be cycle free *by construction*.

Fans of functional programming languages may have noticed many familiar concepts in this section. This is not a coincidence. Functional languages have proven to lead to reliable and performant programs. Meson aims to provide the same power to build definitions by stealing and adapting as many useful tools and techniques as possible.

3.3 Elementary types

String typing, as used in several build systems, is easy to get started with but becomes troublesome fairly quickly. Suppose for example that you want to execute an external command and pass the string `'first second'` as a command line argument. All command shells break arguments at the space character by default, thus that string would be passed as two different arguments, `'first'` and `'second'`. This may be what you want, or it might not be. Perhaps you meant to pass the entire string as a single argument.

There are two different contexts for the string, the *quoted context* and the *unquoted context*. The string itself does not carry any information about how

it should be used. It is the responsibility of the developer to deal with this manually. In practice most people don't, causing their scripts to fail with weird errors when they are fed a file or directory name containing a space. Such names are rare on Unix but common on Windows.

Meson solves these problems by having a rich vocabulary of types. Most of them are the same types you know from other programming languages. This section describes “elementary” data types, whereas the next one is about domain specific object types. The difference between the two is that elementary types are general, whereas domain object types always represent items and elements that are specific to a build system.

3.3.1 Strings

Most data manipulations in build files deal very closely with strings. In Meson strings are defined with single quotes.

```
var = 'some text'
```

This format is convenient for short strings. For large blocks of text, especially those containing new line characters, the triple quoted string is a better choice.

```
var = '''This is the first line of text.
This is the second line.

This is the fourth line after one empty line.
'''
```

As can be seen, the triple quoted string starts with three consecutive single quote characters and continues on until a second set of three consecutive single quotes is encountered.

Strings can be concatenated with the + operator.

```
str1 = 'Hello'
str2 = 'world'
combined = str1 + ', ' + str2
# combined is assigned the string 'Hello, world'.
```

In addition strings can also be joined with the / character. This performs a *path concatenation*. The strings are treated as file system paths and joined together according to the semantic rules governing directory manipulations.

```
dir1 = 'foo' / 'bar'
dir2 = 'foo/' / 'bar'
# both dir1 and dir2 contain foo/bar
```

If the latter string represents an *absolute path*, then the first part is dropped.

```
dir1 = 'foo/bar'
dir2 = '/baz'
dir3 = dir1 / dir2
# Value of dir3 is '/baz'.
```

String objects have a large set of methods. Probably the most used one of these is `format` for creating new strings.

```
msg = 'Value of @0@ is @1@.'.format('variable', 6)
# msg is 'Value of variable is 6.'
```

The `format` method goes through the string looking for substrings of the form `@X@`, where `X` is a number. It will then create and return a new string where those are replaced with the textual representation of the corresponding argument to the method call. In this case `@0@` gets replaced with the first argument, which is `'variable'` and `@1@` gets replaced with the second argument `6`. The formatter is strict: if the string had an substring `@2@` Meson would detect that it is trying to access a nonexisting argument and would immediately halt and exit with a hard error.

The string object contains many other useful methods. A full list with descriptions can be found in Section 17.6.

3.3.2 Integers

In Meson numbers are integers and they behave in the usual way:

```
value = 1 + 2 * 3
# value is 7 rather than 9
# because * has higher precedence than +
```

It is seldom necessary to perform complex arithmetic operations in Meson files. Because of this Meson does not have a concept of rational or decimal numbers, only integers.

3.3.3 Booleans

Boolean variables can hold one of two values, `true` or `false`. They are mostly used for branching in `if` clauses.

Booleans can be combined with the `and`, `or` and `not` operators.

```
v1 = true or false # v1 will have the value "true".
v2 = true and false # v2 will have the value "false".
v3 = not true      # v3 will have the value "false".
```

3.3.4 Arrays

An array is a sequence of zero or more elements that may be of any type. In practice most arrays consist of only one type, most commonly strings, but Meson permits any combination of types.

```
str_arr = ['one', 'two', 'three']
mixed_arr = [1, 'two', 3]
# Both of these are valid.
```

Individual elements of arrays can be accessed with the indexing operator `[]`. Negative indexes count backwards from the end of the array.

```
myarr = ['one', 'two', 'three']
value = myarr[1]
# value contains the string 'two'
value = myarr[-1]
# value contains the string 'three'
```

Indexing out of bounds, that is, with a positive number that points past the last element or with negative number past the beginning, is a hard error.

Like almost all other objects, arrays are immutable. You can not add or remove things to an array once created, you can only create a new one.

```
arr1 = ['one', 'two']
arr2 = arr1 + ['three']
# arr2 is ['one', 'two', 'three']
```

A common operation is to simulate extending an array by creating a new array and assigning it to the same variable.

```
my_arr = ['one', 'two']
my_arr = my_arr + ['three']
```

Because this is done so often, Meson provides a convenient shorthand for this operation, the `+=` operator.

```
my_arr = ['one', 'two']
my_arr += ['three']
```

Note, though, that this does *not* change the old value that was stored in `my_arr`. All places where it was used continue seeing the old value. The only thing this expression does is that from *this point forward in time*, when `my_arr` is used it will contain the new value.

3.3.5 Dictionaries

A dictionary is a data structure consisting of *keys* and corresponding *values*. They similar to dictionaries and hash tables in many programming languages with the limitation that keys must be strings. No other key type is supported and trying to use such a type is a hard error. Thus Meson dictionaries behave almost identical to JSON objects.

```
d = {'key1': 'value1',
     'key2': 'value2'}
```

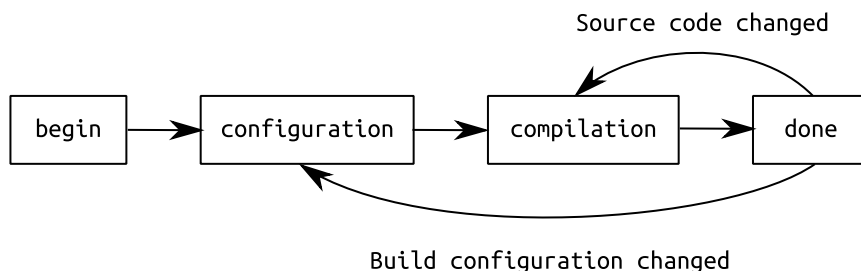
Element access works just like it does for arrays.

```
d = {'key': 'value'}
result = d['key']
# result contains the string 'value'
```

Dictionaries can be combined with the `+` operator in a similar way to arrays.

```
d1 = {'key1': 'value1'}
d2 = {'key2': 'value2'}
d3 = d1 + d2
# d3 contains all entries from d1 and d2.
```

The exceptional case is when both dictionaries to be added contain the same key. In this case the value from the object on the right hand side is used.



31

Figure 3.1: The distinct phases of a build.

```
d1 = {'key': 'vanishes'}
d2 = {'key': 'persists'}
d3 = d1 + d2
# d3['key'] contains the value 'persists'
```

3.3.6 Null type

Many programming languages and scripts have the concept of a *null type*. In C it is called the NULL pointer, in Python it is called `None` and in Ruby it is called `nil`.

Meson does not have such a type because it has been found to be a frequent source of bugs in practice [9]. Instead of a null type you should use a more appropriate type, such as using the empty array `[]` to mean “no entries”.

3.4 Build system phases

Every software project build starts with just a checkout of the source code. It ends with the final built artefacts ready to build and install. Software development is a continuous iterative process between these two extremes as shown in Figure 3.1.

First the build is set up with the `meson` command. This runs the *configuration phase*, in which Meson goes through the build definition files one by one and generates the files needed by the build backend. Then the backend is invoked to run the *compilation phase*. This is when compilers, linkers and other tools are unleashed to build the final artefacts. After this the real work of software development can start. This process is called the *edit-compile-test cycle* because it usually consists of small changes, which are compiled and run until the desired functionality has been implemented.

The configuration and compilation phases have very different behavioural characteristics. Configuration is single threaded and usually fairly fast, even complex projects take under a minute to run. Building is massively parallel but can take a long time. A large project can take hours to build even on high end desktop machines. On the other hand *incremental builds* can be very fast. Changing one file and rebuilding can take only a few seconds even for huge projects. On the other hand configuration always takes roughly the same amount of time.

To keep developers “in the zone” Meson tries to keep incremental builds as fast as possible. This means reconfiguring the project as seldom as possible. Usability, on the other hand, requires that the system detects when it needs a reconfiguration and does so automatically and transparently. Thus changes in the source files themselves do not cause a reconfiguration, but changes in build setup do. This means changes in build definition files and project options.

3.5 Program flow

A Meson build definition consists of *statements* which are executed in sequence. This is known as *imperative programming* and is the most common programming paradigm currently. What makes Meson different from those languages is that a statement, once executed will not be executed again. Or, in other words, the flow of execution only goes forwards. There are no mechanisms for declaring arbitrary loops or custom functions or macros.

In regular programming languages these sorts of limitations could not be made. The end result would not be usable. For a build system these limitations are beneficial, because they provide two major advantages: one theoretical and one practical. The theoretical one is that every Meson build definition is guaranteed to terminate. In computer science whether a computer system will stop for a given input is known as the *halting problem*. For Turing-complete programming languages the halting problem is *undecidable*, meaning it is impossible to prove whether a given program will stop or carry on executing forever. Since Meson build definitions can not loop forever, we can conclude that the definition language is not Turing complete.

The practical advantage is that these limitations make build definitions noticeably simpler to read and reason about, especially when combined with immutable objects. When an object is created, all the information about it *must* be present at the point of construction. Since objects are immutable we know that nothing about them can change after they have been constructed. This reduces the cognitive load on developers when compared to a system that allows build definition data to be mutated in arbitrary ways at any time.

The lack of custom functions also simplifies things. Since all data manipula-

tion is done via the same standard primitive operations, they are the same in all build definitions. This makes it easier to move between different projects, since their overall structure is similar. The obvious downside is that it is not possible to create simple helper macros and functions. In a general programming language this would be unacceptable, of course, but when writing build definitions the problem space is much smaller, so this is a worthwhile tradeoff².

3.5.1 Subdirectories

Having the entire build definition in a single file is simple, but unfortunately it does not scale. Most software projects span multiple directories. Largest such projects can contain hundreds or even thousands of subdirectories. To keep this manageable the build definitions for each subdirectory need to be written in that subdirectory, otherwise things become unmanageable.

Meson supports distributing build definitions across multiple directories natively. Transitioning between different directories happens *explicitly*, only when the build definition executes a subdirectory traversal statement.

```
subdir('dirname')
```

This causes Meson to execute the following steps:

1. Stop executing the current file.
2. Go into the subdirectory specified in the call.
3. Start executing the `meson.build` file in the subdirectory from the beginning.
4. When the build file in the subdirectory has been fully processed, return to the original directory.
5. Resume executing the original build file.

This behaviour can be recursive, any subdirectory may have an arbitrary number of `subdir` calls. This program flow is visualised in Figure 3.2.

From the user point of view the behaviour is roughly similar to having all the code in the subdirectory's file copied in the parent directory's build file. All variables declared in parent directory are visible in the subdirectory and correspondingly all variables and state changes done in the subdirectory become visible in the parent directory once control flow returns. It is common in Meson

²There are people on the Internet who *strongly* disagree with this.

34

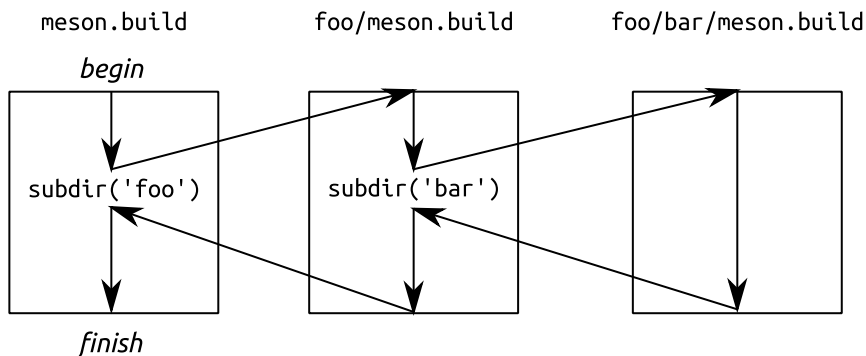


Figure 3.2: How control flows through a Meson build definition spanning multiple directories. Each box represents the contents of a single `meson.build` file.

to define targets such as libraries in one directory, and then build targets, such as tests, using the built library in a different directory.

This is done by first `subdiring` in the directory that holds the library definition:

```
# in top level meson.build
subdir('src')
```

In the `src` subdirectory we define the library target and store it in a variable.

```
# in src/meson.build
lib = shared_library(...)
```

Once control returns to the main build file, we can go in the test directory:

```
# in top level meson.build
subdir('test')
```

where we can finally specify the test executable.

```
# in test/meson.build
executable('test_exe', ...
    link_with: lib) # "lib" was set in "src" subdirectory
```

This works quite nicely but there is a potential source of errors here. If any build file that is processed between subdirectories `src` and `test` reassigns the

`lib` variable, the test executable will see the new value rather than the old one that was set in `src`. This can not be prevented with any technological feature, as it is a policy issue (and sometimes you *do* want to change the value). To avoid mix-ups it is important to give variables unique and descriptive names so they will not be mixed up and reassigned by accident.

3.5.2 Function calls

Function calls are the main vehicle for *getting stuff done* in Meson. They are how you instruct the system to create build and link targets, run system tests and everything else needed in your build. We have already seen many function calls in code samples in this book. Let's now look at how function calls are defined in detail.

There are four main elements in a function call. They are all shown in the following code snippet.

```
return_value = function_name(positional_arg1, positional_arg2,  
    keyword_arg1: value1,  
    keyword_arg2: value2)
```

The most important thing about a function is its name. It uniquely identifies the functionality the function provides. Function names we have already seen include `executable`, and `shared_library`.

Almost all functions need arguments that specify what exactly they should do and how they should get it done. There are two different kinds of function arguments in Meson: *positional arguments* and *keyword arguments*. Positional arguments are the simpler of the two. They consists of entries one after the other separated by commas. A function can take any number of positional arguments, including zero. The `message` function, for example, takes only one argument:

```
message('Some text.')
```

The function call prints the argument to the user's terminal.

```
Message: Some text.
```

The **Message:** prefix is automatically added by Meson so users can easily tell the difference between user messages and Meson status messages.

A positional argument can be any value such as a string, number, array or a dictionary. It can even be a statement. It will be evaluated and its value will be used as the argument. The call to `message` could also have been printed in the following form:

```
message('Some' + ' ' + 'text.')
```

One function that takes a lot of positional arguments is `executable`. After the executable name you can pass all the sources one by one.

```
executable('prog',  
          'file1.c', 'file2.c', 'file3.c', 'file4.c', ...)
```

Function calls like these get fairly tiring to write after a while, which is why Meson does *argument flattening* to make things easier. You can pass positional arguments in different combinations and Meson will figure out what you mean. The simplest case is having the files in an array and then passing that in.

```
arr = ['file1.c', 'file2.c', 'file3.c', 'file4.c']  
executable('prog', arr)
```

Passing some arguments standalone and the rest in an array works too:

```
arr = ['file2.c', 'file3.c', 'file4.c']  
executable('prog', 'file1.c', arr)
```

Splitting the list in two arrays works:

```
arr1 = ['file1.c', 'file2.c']  
arr2 = ['file3.c', 'file4.c']  
executable('prog', arr1, arr2)
```

Fans of deeply nested data structures can even go full Lisp:

```
arr = ['file1.c', ['file2.c', ['file3.c', ['file4.c']]]]  
executable('prog', arr)
```

All of these four code snippets produce the same output: an executable that consists of the four listed source files.

In addition to positional arguments, functions can also take keyword arguments, colloquially known as *kwargs*. They are straightforward key-value pairs and they are listed after all positional arguments. Some functions have mandatory keyword arguments and trying to call them without specifying the given kwarg is a hard error. Usually keyword arguments specify additional features to the function that change the default behaviour. By default build targets are not installed, to specify that we want the target installed, we need to specify the `install` keyword argument and give it the value `true`:

```
executable('prog', 'file.c', install: true)
```

Like positional arguments, the value of a keyword argument can be anything: a string, number, array, a variable or a statement to be evaluated.

Keyword arguments can also be defined dynamically. This is achieved with the special `kwargs` keyword argument. It takes a dictionary as its value and adds all entries in it as keyword arguments to the function. This may seem complicated based on description alone, but becomes clear with an example, where we convert the previous example to use dynamic keyword arguments.

```
kwarg_dict = {'install': true}
executable('prog', 'prog.c',
           kwargs: kwargs_dict)
```

This is identical to having the `install` keyword argument directly in the function call. The only thing to note is that it is illegal to specify a keyword argument both in the `kwargs` dictionary and in the function itself. That is, the following is not permitted:

```
kwarg_dict = {'install': true}
executable('prog', 'prog.c',
           install: true,
           kwargs: kwargs_dict)
```

Meson will exit with an error if it detects duplicate keys, even if the values are the same.

The fourth and final notable thing about functions is their return value. In most cases it is assigned to a variable and then used later. Some functions do not return any values. Trying to use the return value of such a function is an error:

```
# This is an error because message() does not return a value.
var = message('No return value here.')
```

3.5.3 Method calls

Method calls are a way to invoke functionality provided by objects. For example suppose we have an array and want to create a string telling how many elements it has. This can be achieved with the following code.

```
array = [1, 2, 3]
string_template = 'The array has @0@ items.'
length_str = string_template.format(array.length())
# The value of length_str is 'The array has 3 items.'
```

We have used two different method calls to create the string. The first one is the array's `length` method which returns the number of elements. The second is the string object's `format` method, which we use to convert the integer into the final string.

Passing positional and keyword arguments to methods is done in the exact same way as for functions.

3.5.4 If clauses

Out of all all control operations `if` is perhaps the one that requires the least amount of introduction, because it can be found in every programming language and it works in the exact same way in all of them.

```
if x==3
    message('x has the value three.')
elif x>=3
    message('x has a value larger than three.')
else
    message('x is adorable.')
endif
```

The full `if` construct consists of three separate blocks. First there is the mandatory `if`, followed by zero or more `elif` blocks followed by an optional `else` block.

An `if` block starts with the `if` statement. It consists of the `if` keyword followed by an expression which must evaluate to a boolean value. If the value is `true` then Meson will execute all commands until it encounters an `elif` or `else` statement whose contents it will ignore.

If the value in the `if` statement was `false`, the first block is skipped. If the next statement is an `elif` statement it is processed in the exact same way as the `if` statement was. First the expression is evaluated and then the block is either executed or skipped depending on the value. If none of the `if` or `elif` expressions have been true, the flow eventually reaches the `else` block whose contents are executed.

`if` blocks can be nested to form arbitrarily complex decision trees.

```
if condition1
  if condition2
    message('Both conditions 1 and 2 are true.')
  else
    message('Condition 1 is true but condition 2 is not.')
  endif
elif condition2
  message('Condition 1 is false but condition 2 is true.')
else
  message('Both conditions 1 and 2 are false.')
endif
```

If the value of any evaluated branching condition produces something else than a boolean value, Meson will exit with a hard error. There is no implicit conversion from other data types to booleans, so for example the following sample will exit with an error.

```
x = 3
if x    # x is not a boolean, leading to an error here.
  message('x is non-zero.')
else
  message('x is zero.')
endif
```

3.5.5 Ternary operator

A common use of the `if` clause is choosing between two values based on a condition:

```
if condition
  var = value1
else
  var = value2
endif
```

This is such a common thing that Meson provides a shorthand for this: the *ternary operator*:

```
var = condition ? value1 : value2
```

Ternary operators are interesting in that they have a dual nature. When used sparingly they make build declarations simpler, and more readable and

understandable. On the other hand nesting ternary operators tends to produce obtuse and undecipherable definitions where bugs love to hide.

Because of this Meson forbids nesting of ternary operators. If you need more complex decision logic, it needs to be written explicitly with `ifs`.

3.5.6 Foreach clauses

Sometimes you need to perform the same operation on a bunch of items. For example you might want to build many executables with the same options. This is especially common when defining test executables. Suppose we have created an array that lists all the executables's names and source files in a two dimensional array like this.

```
programs = [['prog1', 'source1.c'],
            ['prog2', 'source2.c'],
            ['prog3', 'source3.c'],
            ['prog4', 'source4.c'],
            ]
```

We can then use the `foreach` clause to define all the targets using compact notation.

```
foreach entry: programs
  program_name = entry[0]
  program_source = entry[1]
  executable(program_name, program_source, ...)
endforeach
```

The end result is identical to expanding the loop manually:

```
executable('prog1', 'source1.c', ...)
executable('prog2', 'source2.c', ...)
executable('prog3', 'source3.c', ...)
executable('prog4', 'source4.c', ...)
```

For this simple case both of these declarations are fairly similar but it is a lot simpler to change the executable type in the former, because there is only one `executable` declaration. Thus any changes are guaranteed to be applied to all executables. If the declarations are separate, it is easy for them to go accidentally out of sync, so that some changes are not applied to all invocations. This can lead to nasty bugs. At the very least it is confusing for people who need to understand and edit the build definitions later on in the future.

The `foreach` clause works both for arrays *and* dictionaries. The syntax is slightly different for the latter.


```
dict = {'k1': 'v1',  
        'k2': 'v2'}  
  
foreach key, value: dict  
    message('Processing key @0@, value @1@.'.format(key, value))  
endforeach
```

Running this script will produce either this output:

```
Message: Processing key k1, value v1.  
Message: Processing key k2, value v2.
```

Or this one:

```
Message: Processing key k2, value v2.  
Message: Processing key k1, value v1.
```

The order in which entries are traversed in dictionaries is not specified, so you can't predict which one will be used. You should not write any code that requires dictionary entries to be processed in a specific order. If order is important you should use arrays instead, they are always processed in order.

The observant reader might already have noticed that in Section 3.5 it was stated that Meson does not permit loop constructs because they would cause the language to become Turing complete. Yet this section has gone to great lengths to describe how to write custom loops. This may feel like cheating but actually it is not.

The `foreach` loop is not a free-form looping construct as described in Section 3.5 because it is *fully specified up front*, or more specifically because you can't modify the array when looping over it. Thus if the array has, say, three elements, the loop will go through three iterations and then terminate. If it were possible to change the object being iterated over, an infinite loop could be written. Because Meson objects are immutable, this is impossible and thus the `foreach` loop is safe to use.

3.6 Object types

Elementary types are useful, but defining the entire project's build definitions with them is kind of like building a house out of chopsticks. Sure, you *can* do it and it is an interesting challenge on its own but in practice it's tedious, unproductive and the end result is likely to collapse at the slightest disturbance.

What is needed, then, are higher level constructs that work well together. In Meson these things are called *objects*.

These objects are quite different from objects as understood in common programming languages, such as C++ and Java. There is no inheritance or type hierarchies and users can't define their own classes. Furthermore almost all objects are immutable so you can't change them after they have been created. A more accurate name for Meson objects would be something like *opaque containers of domain-specific information*, but it does not roll off the tongue quite as smoothly as the word object.

There are three basic things you can do with objects. First you need to create one. For example, here is how you'd create a shared library object:

```
shlib = shared_library('hello', 'hello.c')
```

The second, and perhaps the most common, thing is to use the object as an argument to other functions and methods. For libraries this usually means linking against them.

```
exe = executable('printer', 'printer.c',  
                link_with: shlib)
```

The third thing to do is calling methods on the object. A shared library object does not have many interesting methods, but suppose we wanted to print the full path of the object to the screen. It could be achieved with the following code.

```
message('The full path to shlib is:' + shlib.full_path())
```

3.7 Disablers

When a project gets bigger and more complex, one often ends up with a need to disable parts of it. This can be either temporary for debugging purposes, or permanent that depends on e.g. optional dependencies. The most common way to do this is with `if/else` logic, but writing branches everywhere can get tedious. Meson provides a helper primitive for this called a *disabler object*.

The basic concept is simple. Whenever a disabler object is passed as an argument to a function or method call, the function is ignored and its return value is a disabler object. This is roughly similar to how *not a numbers* (NaNs) behave in floating point math. Thus the values of all of the following declarations will be a disabler object.

```
d1 = disabler()
d2 = executable('name', d1, ...)
d3 = 1 + d1
d4 = 'string' + d1
```

The same happens with branching logic, but there is an additional thing to note. A disabler skips *every* branch.

```
d = disabler()

if d
    # This branch is skipped
else
    # This branch is also skipped.
endif
```

Disablers are greedy so one needs to be careful when using them. They are most useful for disabling individual components such as plugins based on whether their dependencies are found or not. They don't work in the case where you build one library or executable from many parts, because if any of the subcomponents is a disabler, the entire target will be gone, which is usually not the desired outcome.

There are three exceptions to the rule that disablers short circuit evaluation. They are all functionality for setting and inspecting the contents of variables.

```
d = disabler()
is_found = d.is_found() # Returns false
set_variable('d2', d)   # d2 will be set to a disabler
is_d = is_disabler(d)   # Returns true
```


Chapter 4

Building blocks of a software project

Every build system can be reduced to a pipeline architecture consisting of two different elements: transformation blocks and connections between them. In this way they are similar to Unix shell pipelines. A shell command to sort the lines in a file and print the 10 first unique lines looks like this:

```
$ cat file.txt | sort | uniq | head -10
```

Each of these commands works in isolation. It does not need to know anything about the other parts of the pipeline or even that there is a pipeline. The job of the programmer is to manage *data flow*, specifying what data goes from one command to another and the operational details of each block. Those are specified with command line arguments to the individual commands. The same applies to defining a project's build setup.

This is where the similarities end. When defining a shell pipeline you have a rich vocabulary of types, a typical Unix OS has hundreds of commands to choose from. The data flow in shell pipelines is linear. It goes from one program to another in a strict sequence. Build systems are the opposite of this. There are only a few building blocks. In fact it is possible to define fairly large software projects using only one operation: building an executable. Most projects consist of libraries as well as executables for convenience. The data flow model, on the other hand, is complex. The output of one block can be used as an input to pretty much any other block. It can even be used as input multiple times in different blocks. The data flow diagram in build systems is not linear, but instead it is an arbitrary graph.

Most build systems, Meson included, limit the graph to being a *directed acyclic graph*. This kind of a graph does not have *circular dependencies*, which

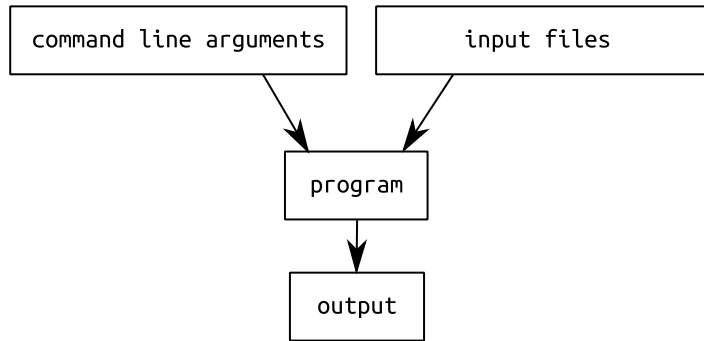


Figure 4.1: The elementary transformation operation of a build system.

are widely accepted to be a sign of poor design. A circular dependency happens when the output of block *A* is used as input to block *B*, and the output of block *B* is *also* used as input to block *A*. Running this leads to an eternal loop and this is why these constructs are prohibited.

4.1 The elementary operations

The basic operating block of the build graph can be seen in Figure 4.1. A graph for Unix shell would look identical. The figure basically describes executing an arbitrary command line program. Every other operation is just a special case of this operation. The common case is not even particularly interesting because it is so general. Things only get interesting once we add different types of operations, which is done via limitations.

The most common elemental operation in a build system converts source code into a library or an executable. This is called the *compile and link* step. As you can probably tell from the name it internally consists of two different steps. Since those are implementation details and do not affect the build graph, we can treat them as a single operation in this chapter. The diagram for this operation can be seen in Figure 4.2.

Let's start with the output, since it is identical to the general case. The output file can be either a library or an executable. The difference between the two is that executables can be run but not linked whereas libraries can not be run but they can be linked into other targets¹.

¹Neither of these statements is *strictly speaking* true, but that gets into very deep, very black magic.

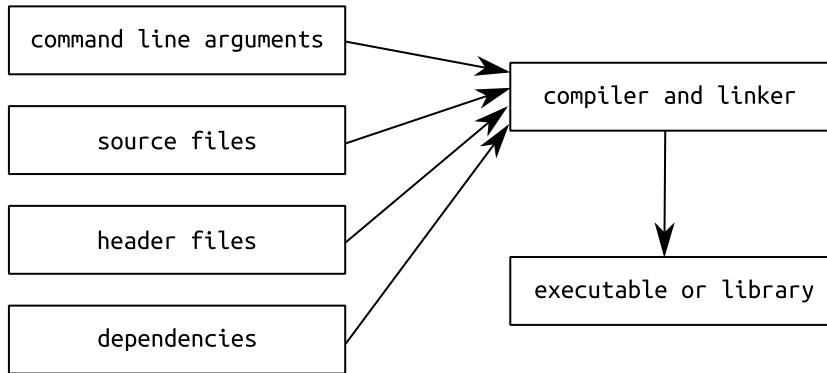


Figure 4.2: The basic build operation converts source files and extra data to an executable or a library.

Inputs are where things start to deviate from the common case. There are three distinct input file types: headers, sources and dependencies, which in practice means libraries. The first two are used during the compilation step and the last one during linking. In addition to files, dependencies may also contain command line arguments. They must also be used to build the target successfully. All of this will be handled by the build system automatically once the developer has set up all dependencies correctly.

The output library of one target can be used as a dependency input in another. This may produce yet another library that can be used in further builds. The dependency library need not be self built, it can also come from the toolchain. On most programming languages at least the standard library is provided by the system, so programs don't need to rebuild it from scratch. On unixlike platforms using system provided libraries is both convenient and common. Other platforms do not have a concept of system dependencies and thus require users to do more work. Meson supports both of these workflows natively.

4.2 Advanced build cases

Compilation and linking make up 95% of all the work for 95% of all projects. Sometimes it is not sufficient, though, and we need to generate something else, such as documentation. These are standalone things that do not affect the compilation phase. Where things get really interesting is when we realise that all inputs listed in the flow diagrams can be generated during the build. Meson

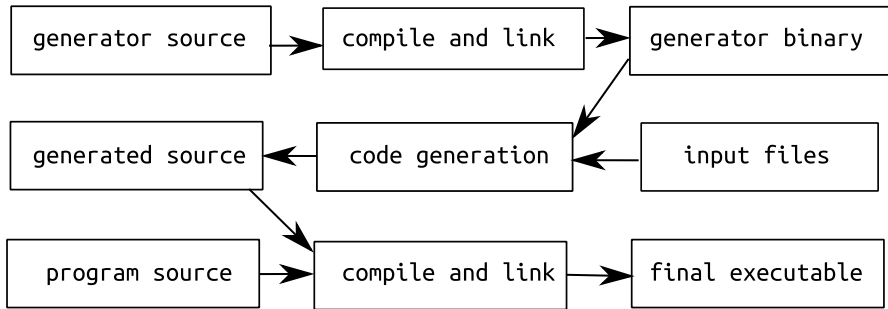


Figure 4.3: A pipeline that builds a code generator, uses it to generate source code and compiles the result.

has two limitations, command line flags and compilers can not be created on the fly, they have to be specified up-front in build files. Everything else can be generated. This limitation is not theoretical but an implementation choice. It is possible to create a build system that is fully dynamic in this way but at the time of writing nobody has yet created one.

Generating source code from some sort of definition file is a well established discipline. It is typically used to handle serialisation or network protocols from a higher level description language called a *interface description language*. The main advantage of this is that such formats are easier for humans to write and maintain than code that does equivalent work in the language itself. It also allows one to generate code for multiple different languages from a single description file.

The source code generator (often called a “compiler” but we’re not using that term to avoid ambiguity with source code compilers) can either be found on the system or it can be built as part of the build. If it is self built, the chain from plain sources to a fully built executable is fairly complex. One such pipeline can be seen in Figure 4.3.

The diagram illustrates nicely all the things discussed thus far. Mainly, that the source of complexity comes from the way the individual blocks are connected together. Each individual operation is simple and it can be analysed in isolation from the rest of the build graph. The flow of data only goes forwards, meaning we know things that must happen strictly before and after each individual task. A graph with these properties can grow almost arbitrarily large and complex and it can still be analysed and understood without superhuman effort.

4.3 Generating data

49

Executables and libraries make up most of the build system's work but not of the output. A project usually needs a bunch of generated data files such as man pages, documentation and resource files. These may require considerable work. In computer games it is typical to have an *asset pipeline* that takes original art and converts, compresses and packs it into the final form used by the game executable.

Even though the actual generation process may be heavy and involved, to a build system it is simpler than compilation. Every data generation system can be reduced to the primitive operation in Figure 4.1: some files go in, a command is executed and then other files come out.

4.4 Defining the graph in Meson

Building libraries and executables is most of the work a build system has to do. It is not surprising, then, that defining libraries and executables, commonly known as *build targets*, takes most of developer time when writing a build system for a software project. Let's get started with the first executable example from Chapter 1.

```
project('simple demo', 'cpp')
executable('hello', 'hello.cpp')
```

Let's split this program into two parts. We'll create a helper library that does the actual printing and an executable that uses it. First we need to write a header file for the library called `greeter.h`

```
#pragma once

void print_greeting();
```

The source file `greeter.cpp` holds the implementation.

```
#include "greeter.h"
#include <iostream>

void print_greeting() {
    std::cout << "Hello, world!\n";
}
```

This simplifies `hello.cpp` to just calling the helper function.

50

```
#include<greeter.h>

int main(int argc, char **argv) {
    print_greeting();
    return 0;
}
```

The source files have gotten (arguably) simpler, but the build definition gets more complex as it has more to do. Instead of building a single executable, we now need to build a library, an executable and make sure that the latter links against the former. For simplicity we are going to use a static library.

All build definitions are still contained in a single `meson.build` file, but we're going to go through it in segments. First we define the project and build the library.

```
project('library demo', 'cpp')
greeter_lib = static_library('greeter', 'greeter.cpp')
```

The only thing of note in this snippet is that we store the return value of the `static_library` call into a variable called `greeter_lib`. This is needed so we can refer to the library later. If the return value is not stored, it is *impossible* to get a hold of the library object ever again.

Now that we have a library object, we can tell Meson to link the executable against it.

```
executable('hello', 'hello.cpp',
    link_with: greeter_lib)
```

This is fairly anticlimactic, but this is where Meson differs from most other build systems which tend to specify this same information with a string. As was discussed in Section 3.2, this ensures that all build graphs are acyclic.

4.5 Splitting the project to multiple directories

Most new projects start with a bunch of files in a single directory. This is a perfectly valid approach and can scale to surprisingly large projects. A project providing a few command line applications might not need any more structure than this. Most projects have a tendency to grow larger and larger and eventually splitting the code base into logical components starts to make sense. The most common approach is to put each component and all its source files in their own directories.

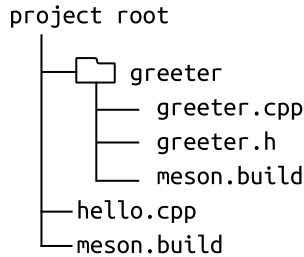


Figure 4.4: A possible directory layout of a project with a standalone library component.

Moving targets to different directories takes only a small amount of effort in Meson. To see just how much, we’re going to move the greeter library to its own subdirectory called **greeter**. The resulting directory layout can be seen in Figure 4.4. All source files retain their contents unchanged. Only build definition files need to be changed.

The build file in the **greeter** subdirectory has the **static_library** declaration and nothing else. It is identical to the definition that was used before the split.

The top level build file becomes simpler.

```
project('split demo', 'cpp')
subdir('greeter')
executable('hello', 'hello.cpp',
    include_directories: 'greeter',
    link_with: greeter_lib)
```

The first change is the **subdir** call that tells Meson to go and process the **greeter** subdirectory. After it has finished, Meson will continue processing the next line that contains the executable definition. It has one new keyword argument, **include_directories**. It tells Meson to add the given directory to the set of *header search directories* for the given target. Without this declaration the source code would fail to compile. The main file’s include declaration `#include<greeting.h>` would fail, because the compiler would not be able to locate **greeting.h**.

It might seem confusing at first sight that linking can use the **greeter_lib** variable even though it is not defined anywhere in the current file. All variables set in subdirs persist and are accessible in their parents after subdir processing is done. In computer science terms Meson does not have *scoped variables*, only

global ones. Global variables are strongly discouraged in general purpose programming languages, for very good reasons. But since Meson is a declaration language, this design is suitable, because it makes variables' life cycles simple.

4.6 Target properties

The commands that define build targets have a substantial number of customisation points. As projects grow it fairly quickly becomes necessary to set these to non-standard values. Fortunately here, as in most cases, you usually only need to tweak only a few of the available options. Out of these the most common is setting custom compiler flags.

To see how this is done, suppose we want to add a debugging tool to the greeter library. It is used by writing a function call `TRACE()` to the source code. This will cause the system to print out a trace message that consists of the current function's name, source file and line number.

The function that does the actual printing looks like this:

```
void print_trace(const char *function_name,
                const char *file_name,
                const int line_num) {
    std::cout << "TRACE: "
              << function_name
              << ' '
              << file_name
              << ':'
              << line_num
              << '\n';
}
```

We don't want these messages written when the program is being run normally, only when it is built in a special mode that enables trace debugging. In C-like languages this is done with the preprocessor by defining the `TRACE` function call to either call the `trace` above or do nothing based on the value of a given token.

```
#ifdef GREETING_TRACE
#define TRACE() print_trace(__PRETTY_FUNCTION__, \
                           __FILE__, __LINE__)
#else
#define TRACE()
#endif
```

If your C-preprocessor-fu is not that strong, this code defines the `TRACE` macro depending on the value of the preprocessor variable `GREETING_TRACE`. It is usually a good idea to prefix preprocessor option names like these with a unique string that refers to the name of the current project. This avoids token name collisions. If this variable is set, then debug output is printed and if it is not set, then the `TRACE` function call vanishes without a — well — trace.

The last thing we need is a way to set `GREETING_TRACE` from the build definition file. This is done by adding a new command line option to the greeter build target using the `cpp_args` keyword argument for C++. Similarly the compilation arguments for C would be set with the `c_args` keyword argument, Java arguments with `java_args` and so on.

```
greeter_lib = static_library('greeter',  
    'greeter.cpp',  
    cpp_args: ['-DGREETING_TRACE'],  
)
```

All of these settings are *per target* only. That is, they are only used when building this particular target. They won't be used anywhere else, including any targets that link with this library. Some build systems behave differently in this case and apply some (but usually not all) properties transitively. This is usually not a problem but people accustomed to the transitive behaviour might be surprised by this.

All the plumbing is now set up and we can add trace statements to the code.

```
void print_greeting() {  
    TRACE();  
    std::cout << "Hello, world.\n";  
    TRACE();  
}
```

After a build we can run the program and see the trace statements as expected.

```
$ ninja  
[4/4] Linking target hello.  
$ ./hello  
TRACE: void print_greeting() ../greeter/greeter.cpp:24  
Hello, world.  
TRACE: void print_greeting() ../greeter/greeter.cpp:26
```

Tracing can be disabled by removing or commenting out the line containing `cpp_args` keyword argument from the build definition file. This is a coarse

approach but it works for features that change rarely. Later in this book we are going to examine ways to make switching between the two modes smoother.

Chapter 5

External dependencies

John Donne was an English poet who lived in the 1600s. His best known poem deals with interactions between humans [4].

*No man is an Island, intire of it selfe;
every man is a peece of the Continent, a part of the maine*

There are many ways to interpret this phrase, one of which is that while humans can work in isolation, their achievements and ideas only truly come to life when interacting with their fellow humans. The same can be said of software systems. Converting Donne's quote to contemporary software engineering, one might end up with something like the following.

`No software program is an island, with the exception of a
group of highly specialised programs, mostly embedded firmware
that do not communicate with other systems, and which can
usually be ignored in order to make the general case easier
to reason about.`

This demonstrates why engineers rarely make good poets.

5.1 What is a dependency?

Dependencies are an integral part of modern software development but there is not an agreed-upon definition of what constitutes a dependency. For the purposes of this book we shall define that a dependency is *any piece of functionality that does not come from the current project being built*. This definition is fairly broad and there are setups where this does not hold but it covers the common use cases fairly well.

Within this definition there are many different types of dependencies. The most elementary dependency is the *standard library* of the programming language in use. Some languages, such as Python and Java, have an extensive standard library. In these languages it is possible to create large and complex programs using no other dependency than the standard library. Other languages, such as C, go the opposite way and have minimal standard libraries. It is even possible to compile programs without any standard library components.

When talking about dependencies, the most common thing people think of is the *prebuilt third party library*. In the C family of languages this means a library (either shared or static) of code and one or more header files that describe the library's interface, also known as the Application Programming Interface or API. These libraries can either be provided by the system or a standalone dependency manager tool. A real-world example of this would be building a GUI application on Linux. The system typically provides the GTK widget toolkit as a system library. The application needs to compile and link against it to make use of the functionality. This chapter focuses mostly on these kinds of dependencies.

Some dependencies are not provided as libraries but instead as source code that projects are expected to copy inside their own projects. This is an uncommon approach for dependencies, because keeping the embedded copy up to date is a fair bit of work.

Some languages have unique dependency types, such as C++ which has the concept of *header only libraries* that are like regular dependencies except that there is no library component. Instead all the code is in the header and it is compiled inside the targets using it. From the user point of view there is no difference between these two types. Source files in a project include the headers they need and the build system will link the output with a library if needed.

Not all dependencies are code. Tools and executables also count as dependencies, because without them the project can not be built. Generators that convert description languages to source code are the most common example, but there are many other types of executable dependencies. Documentation generators such as Doxygen and Gtk-Doc, static analysers and other such tools are all useful dependencies in software development.

5.2 Finding and using dependencies

Using dependencies is a two phase task. First you need to tell Meson to find your dependency. We'll use the Zlib compression library as an example.

```
zlib_dep = dependency('zlib')
```

The return value `zlib_dep` is a *dependency object* that encapsulates every-

thing needed to use the given dependency. If the dependency can not be found, Meson will exit with a hard error. Thus we know that the result object is always valid.

The second phase is using the dependency when defining a build target.

```
exe = executable('program', 'program.c',  
                dependencies: zlib_dep)
```

This tells Meson to set up the build of `program` so that it can use the `zlib` dependency. Usually this means adding compiler flags that add the dependency's headers in the compiler's search path and linker flags that link the executable against the `Zlib` library.

The resulting compiler command line might look something like this:

```
$ gcc -c -o program.o -g -I/usr/include/zlib program.c
```

The `-I` argument adds the header directory to the search path¹. The corresponding link command might look like the following:

```
$ gcc -o program program.o -lzlib
```

The only thing added is the linker argument `-lzlib`. Its semantics are roughly *look up a file named `libzlib.so` or `libzlib.a` in the system default library directories and link it to the program*. If the library had been at a custom location, we would have needed to add it to the library directory list with the command line argument `-L/path/to/directory`. This is the equivalent of specifying a `-I` argument to the compiler.

5.2.1 Modular dependencies

Most external dependencies provide only one item, usually a library. Some are large conglomerations that provide components that can be used independently of each other. Perhaps the most known example is the `Qt` framework [15]. It has functionality ranging from basic container types to graphical user interface widgets, networking code and even a fully fledged web browser component.

Choosing which subcomponents should be used in a dependency is done with the `modules` keyword argument. A CLI application using `Qt` would be defined like this:

¹On many unix platforms `Zlib` does not need an include argument since its header is in the system global include directory. However we have used one here for expository purposes.

58

```
qt5core_dep = dependency('qt5', modules: 'Core')
```

The module keyword can have more than one entry. A typical dependency declaration for a graphical Qt application that also uses the network and printing requires only slightly more typing.

```
qt5_dep = dependency('qt5', modules: ['Widgets',
                                       'Network',
                                       'PrintSupport'])
```

There is no formal specification on which dependencies have modules, what those modules are and how they should be used. Each dependency type gets to decide that on a case-by-case basis. At the time of writing the only dependencies that provide modules are **qt5**, **qt4** and **boost** with the module names mapping directly to the underlying projects' components.

5.3 Dependency provider backends

Now that we know how dependencies are used, a reasonable follow-up question is how does Meson find the dependencies. This information is especially useful for debugging why dependencies are not found even though they should be. Meson does not look up anything by itself, instead it shifts that task to one of many different *dependency provider programs*. Meson tries to select the correct backend to use automatically but this can be overridden with the **method** keyword argument.

```
d = dependency('depname', method: 'lookup_method_name')
```

The function supports the following lookup methods.

5.3.1 Pkg-config dependencies

The most used dependency program is called *Pkg-config* [8], which is the de facto standard of providing dependency information on Linux and other free and open source operating systems and which also works on other platforms like Windows. This is the default dependency lookup method of Meson. Inspecting what dependencies are available from pkg-config on the current system is easy with the command line tool. Here is a command that prints 10 random dependencies that are installed on the current system:

```
$ pkg-config --list-all/tail
python-3.6      Python - Python library
menuw          menuw - ncurses 6.1 add-on library
gtk+-3.0       GTK+ - GTK+ Graphical UI Library
ncurses        ncurses - ncurses 6.1 library
Qt5Concurrent  Qt5 Concurrent - Qt Concurrent module
libidn2        libidn2 - Library for IDNA2008 and TR46
gio-2.0        GIO - glib I/O library
gapi-2.0       GAPI - GObject .NET API Wrapping Tool
xdmcp          Xdmcp - X Display Manager Control Protocol
renderproto    RenderProto - Render extension headers
```

Inspection what command line arguments each dependency adds can be done with the `--cflags` argument.

```
$ pkg-config --cflags ncurses
-D_GNU_SOURCE -D_DEFAULT_SOURCE
```

On this system ncurses does not require any custom header include paths, but it does require two preprocessor definitions. Linker arguments can be obtained in the same way.

```
$ pkg-config --libs ncurses
-lncurses -ltinfo
```

Here we see that using ncurses requires linking against two different libraries.

In addition to using pkg-config dependencies, Meson also provides helper functionality to create pkg-config output files. See Section 20.7 for details.

5.3.2 CMake dependencies

CMake [3] is a popular build system that also has its own dependency provider mechanism. It also provides a way to provide dependencies in similar way to pkg-config but the mechanism is only for CMake's own use. It is not meant for general usage.

Meson can use CMake to extract this dependency information and use CMake dependencies as if they were native. As can be expected this is not as reliable as something like pkg-config.

5.3.3 Qmake dependencies

Qmake is a build tool provided by Qt. It provides a way to get dependencies about the Qt installation much like pkg-config. Qmake only provides information

about Qt, other packages don't integrate with it. Qt does provide dependency information also via pkg-config files on some platforms.

60

5.3.4 Custom script dependencies

Some projects do not trust third party tools such as pkg-config but instead provide their own standalone scripts to provide dependency info. They behave mostly identical to Pkg-config but you just need to know to call, for example, `cups-config` rather than `pkg-config`. Meson maintains a list of these dependencies and tries to use the custom scripts whenever possible.

Some projects provide both dependency types. For example SDL2 provides both pkg-config files as well as its own `sd12-config` script. In these cases Meson will default to using pkg-config.

5.3.5 System dependencies

Some dependencies are provided by the operating system in a special way. A common example is OpenGL. These dependencies are platform-dependent.

5.3.6 Apple framework dependencies

Both macOS and iOS have their own dependency mechanism that is based on *frameworks*. These are bundles of libraries and headers that need to be installed on a specific location on the device. The Meson name for this dependency method is `appleframeworks` and it looks up the given framework by name from the system directories.

5.4 Executable dependencies

Finding executables to run from the system is done with the `find_program` command.

```
prog = find_program('programe')
```

Like with dependencies, if the program is not found, Meson will exit with a hard error. Programs found in this way can be either invoked immediately with `run_command` or used in custom and run targets.

Programs are looked up in directories defined in the `PATH` environment variable. In addition the lookup mechanism will, as a special case, look up programs (usually scripts) that are in the current source directory. On unix platforms whether a command can be executed is based on how the executable bit is

set on the file. Meson assumes that all files marked as executable are directly runnable. On Windows the lookup is trickier. By default only files with the extension `.exe`, `.com` or `.bat` can be run, but most scripts do not have those extensions. For example Python scripts either have the extension `.py` or have no extension at all. In these cases Meson will examine the contents of the script file to see if it starts with the Unix shebang line:

```
#!/usr/bin/env python3
```

This tells the Unix shell interpreter that this script should be run with the `python3` command. Meson will see if Python 3 is available on the current system and if it is, transparently creates a program object that runs the script via the given interpreter. Since Meson is implemented in Python 3, a Python interpreter is guaranteed to always be available. Thus all scripts written in Python 3 can be used on any platform with the same Meson snippet:

```
script_prog = find_program('somescript.py')
```

5.5 Dependencies that don't provide any dependency files

Most libraries in common use provide dependency information, usually via `pkg-config`. Sadly this is not always the case, especially for low level and “vintage” libraries. In these cases you need to introspect the build environment directly. This is done with functionality provided by the compiler object. As most dependencies are libraries, the most direct way is to look it up directly by name.

```
cc = meson.get_compiler('c')
lib_dep = cc.find_library('something')
```

The output `lib_dep` is an object that can be used like a dependency.

```
exe = executable('program', 'program.c',
                 dependencies: lib_dep)
```

Sometimes you might need to look up headers. For example in C++ it is common to provide functionality as header-only libraries. The lookup code is roughly similar:

```
if cc.has_header('functionality.h')
    # The header was found.
else
    # The header was not found.
```

```
# Either fallback to something else or exit.
endif
```

62

Since this would be too easy, some libraries are needed on some platforms but not on others. The C math library is a well known example. It is part of the standard library on every platform except Linux. Thus we do need to look it up on Linux but *not* on other platforms.

```
m_dep = cc.find_library('m', required: false)
exe = executable('program', 'program.c',
                 dependencies: m_dep)
```

On platforms that don't have a standalone math library, the lookup fails and returns a not found object. Build targets ignore those so this code, while a bit tedious, works on all platforms. Other libraries that behave like this include `dl` and `rt`.

As a final exception we have threading support. It works differently on different platforms and there are many ways to set it up that almost work. For this case Meson provides a pseudo `threads` dependency:

```
thread_dep = dependency('threads')
exe = executable('program', 'program.c',
                 dependencies: thread_dep)
```

Chapter 6

Subprojects and internal dependencies

Chapter 5 discussed how to use external dependencies provided by the system. They are highly convenient and easy to use but there are many cases where system dependencies are not an option. The most common reason is that not all dependencies are available as system dependencies. Operating systems update fairly infrequently, so it is possible that you need a newer version of the dependency than is currently provided by the system.

Many operating systems do not provide any third party dependencies. The only thing you get are core system functionality. There are many third party dependency provider systems for these cases, but using one is always an extra hassle. It would be convenient to be able to build your dependencies transparently at the same time as the actual project. Meson provides for this using a mechanism called *subprojects*.

6.1 Subproject basics and layout

A subproject is simply a way to build one Meson project in an isolated sandbox inside a different project while using the same options and settings for both. There is always one master project, which can have zero or more subprojects. Subprojects may not be nested meaning they can not have subprojects of their own. They can use other subprojects in the master project. A visualisation of the layout can be seen in Figure 6.1.

Not being able to have subprojects within subprojects may seem limiting (and it is), but there are very good reasons for this design decision. The main one is reliability, meaning that within one project there *must* be only one version of any dependency. Linking more than one version of a library (even accidentally)

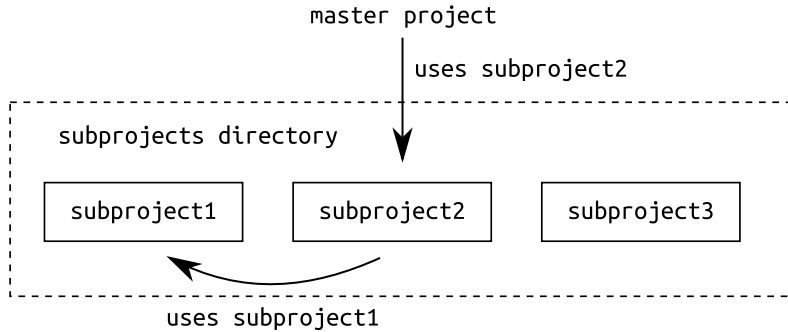


Figure 6.1: There is only one level of subprojects. If a subproject tries to use a different subproject, it gets the one belonging to the master project, not its own embedded copy.

in the same target is almost guaranteed to be *undefined behaviour* with the nastiest of outcomes. The program might work fine for a long time but break mysteriously in production or on a slightly different machine. This can not be reliably prevented.

The other main advantage of this setup is that choosing which version of any given dependency is unambiguous. If there are many different copies embedded around the project tree, it is not clear which one of them actually got used. Furthermore, even innocuous changes might change which version is used. This can cause a project that was working to fail, either by not building at all or failing tests or just plain misbehaving at runtime.

6.2 Using subprojects

There are two different ways of using a subproject. The first is the `subproject()` function call. The second one is an indirect method that will be described later in this chapter as it requires more background knowledge. The basic invocation looks like this:

```
sp_obj = subproject('projname')
```

This call causes several fairly complex things to happen behind the scenes. First of all Meson will suspend the state of the current project. It will then look up the specified subproject, which must reside in `subprojects/projname` at the project's source root. It will then create a new, clean state for the interpreter that runs the builds.

It will then change to the subproject directory and execute the build definition there. From the subproject's point of view the invocation is identical to if it was being run directly. A project can query if it is being used as a subproject, but most projects don't need to know or care about that.

Once the subproject has finished, Meson will take it and store its state in a subproject object, which is then returned from the `subproject` function call. The master project will then continue executing. All things defined in the subproject will be built and used: its tests will be run when executing the test target and its files are installed when running the install target. If any step in subproject configuration fails, Meson will exit with a hard error. This can be avoided with the `required` keyword:

```
sp_obj = subproject('nonexisting', required: false)
```

In this case Meson will return an empty subproject object. The object has a `found` method that can be used to check if the project was found.

Running a subproject by itself is not that helpful. Because all projects are isolated from each other via sandboxing, they can't really communicate directly. There is no way for subprojects to query things from the parent project, but the subproject does provide a way to reach inside the subproject. This is done with the `get_variable` method. Suppose that you want to link against a library built in a subproject. For this you need two things, an include directory for headers and the library itself. They can be defined in the subproject with these lines of code somewhere in the project definition:

```
inc = include_directories(...)
lib = shared_library(...)
```

To build against this library the master project would need to issue these commands:

```
sp = subproject('spname')
sp_inc = sp.get_variable('inc')
sp_lib = sp.get_variable('lib')
executable(...,
              include_directories: sp_inc,
              link_with: sp_lib)
```

The `get_variable` method hoists variables from the subproject to the current project. Since variables holding include directories and build targets can be used in any directory, they will also work when used in the parent project. Meson takes care of expanding all paths correctly.

This direct approach works and is simple but on the other hand it is fairly tedious with a lot of manual work and has other downsides as well. For example if the subproject changes the names of the variables where it stores its dependency information, projects using it will no longer work.

6.3 Internal dependencies

To fix these problems and to make the entire process of using subprojects smoother, Meson allows projects to define their own dependency objects with the `declare_dependency` function. Dependency objects defined in this way behave identically to external dependency objects obtained with the `dependency` function.

Adding a dependency object to the subproject discussed earlier requires only one more line of code:

```
inc = include_directories(...)
lib = shared_library(...)
main_dep = declare_dependency(include_directories: inc,
                              link_with: lib)
```

The name of the dependency variable is arbitrary, but `projectname_dep` is the recommended naming scheme. Linking against the subproject library is now simpler:

```
sp = subproject('spname')
sp_dep = sp.get_variable('main_dep')
executable(...,
              dependencies: sp_dep)
```

Internal dependency objects are useful even without subprojects. Every time you need to build a target which uses a self built library and its headers, you might consider encapsulating the library in a dependency object instead. A typical case is tests. There may be tens of test executables for one library, and dependency objects make linking a lot less cumbersome.

6.4 Combining subprojects and internal dependencies

Thus far we have seen how to use an external system dependency and how to use an internal, self built dependency. This raises the obvious questions: which one of these should be used? The non-obvious answer is that you don't have to

choose only one. A more elaborate answer would be that if the dependency you need is available on the system, you should use that and if it is not, then you can rebuild your own version as a subproject.

An implementation in Meson code would look something like this:

```
foo_dep = dependency('foo', version: '>1.2.0', required: false)
if not foo_dep.found()
    foo_sp = subproject('foo', version: '>1.2.0')
    foo_dep = foo_sp.get_variable('foo_dep')
endif

executable(..., dependencies: foo_dep)
```

This works nicely, but it is messy. There is a lot of typing and things like the version requirement are duplicated. Since this is such a common operation, Meson provides a shorthand syntax for this.

```
foo_dep = dependency('foo',
                    version: '>1.2.0',
                    fallback: ['foo', 'foo_dep'])
```

This is the indirect way to invoke a subproject that was mentioned earlier. The only thing that has been added is the `fallback` keyword argument. Its value is an array with exactly two string arguments. The first one is the name of the subproject to invoke. The second is the variable name holding the main dependency object. The return value can be an internal or an external dependency object, but the project does not have to care. It can just use the object it was given.

This change simplifies and reduces the amount of code needed to use dependencies. More importantly, it shifts the decision on which dependency to use *away from the build definition* and makes the build definition more declarative. Each project does not specify the steps to take to use a dependency. It merely states its requirements for dependencies and lets someone else take care of the lookup and decision logic. This allows Meson to, for example, add a completely new dependency lookup mechanism and have it automatically work on *all* existing Meson projects without needing to change their build definitions at all.

6.5 Overriding executable lookup

Some dependencies provide programs to run in addition to libraries. This is most common for projects that provide source code generators. The concept

of source code generation will be explained in detail in Chapter 11, but for the purposes of this chapter we will only concentrate on executable lookup.

Picking between system and subproject provided executable manually is simple:

```
if <use subproject version>
    sp = subproject(...)
    dep = subproject.get_variable('dep')
    exe = subproject.get_variable('exe')
else
    dep = dependency(...)
    exe = find_program(...)
endif
```

This is a lot of typing so the obvious solution to this would be to use the same fallback mechanism to `find_program` as is used in dependencies. *This will not work.*

```
dep = dependency(..., fallback: ['sp', 'dep'])
# This does NOT work!
exe = find_program(..., fallback: ['sp', 'exe'])
```

This is because `find_program` does not support fallbacks. The reason for this is that executables work differently from library dependencies. It is entirely possible to have a dependency executable installed on the system even though the corresponding libraries are not. In that case the above declaration would fail, because it would use the library dependency from the subproject but the executable from the system. There is no guarantee that this combination will work. At best it will cause a build failure and at worst creates programs that have subtle but mysterious bugs.

Because of these reasons Meson provides a different way of getting executables that is based on *overriding*. Any project can call the executable override function like this:

```
exe = ...
meson.override_find_program('exename', exe)
```

This changes Meson's executable lookup logic so that this code:

```
exe = find_program('exename')
```

does not try to look up the executable from the system but instead immediately returns `exe` as specified in the override command. The override value can be

any executable type, that is, an executable or a custom target or the output of a different `find_program` call.

This works nicely and transparently for most projects but it adds one minor limitation. If the overridden value is an executable target, then the result can not be used in a `run_command`. It must execute the program immediately at configure time, but executables are not built yet so they can't be run. This is only a limitation if you need to get some information out of the program during configure time. This is rarely needed, as the most common thing to look up is the program's version number. The recommended solution is that it should be looked up in the dependency object instead. It has native support for versions so it is more reliable than string matching on an executable's output.

As an additional safety feature Meson will ensure that all lookups for a given executable name are the same across all subprojects. Thus any executable name that has been successfully found can not be overridden any more. Similarly trying to override an already overridden executable name is also prohibited. Trying to do either of these will cause Meson to exit with a hard error.

Chapter 7

Configuring the project

A unique property of software is its near infinite flexibility. Almost any code can be adapted to work on multiple different platforms and to interface with many different libraries, frameworks and toolchains. Since all these systems work in slightly (and sometimes not so slightly) different ways, a certain amount of *configurability* becomes necessary. There are many drivers for configurability, for example a multimedia processing library might need to have a way to disable all algorithms that might infringe on patents. Users who either have a patent license or who live in a country where software is not patentable can then enable those algorithms with a command line switch.

Regardless of the reasons for any configuration it breaks down to roughly two different steps. The first one is *introspection* where the build system determines which features have been requested and what features are available on the current system. This info allows it to decide what sort of configuration should be used. The second step is to *generate the configuration*, which usually consists of defining compiler flags and writing configuration files. Once this is done, the actual build process can start.

7.1 Simple approaches to configuration

Some configuration choices are simple. If they match your needs then you should consider using them if you can. In this section we look at some simple approaches for project configuration and also outline their limitations and reasons you might use the more complex configuration mechanisms listed later in this chapter.

7.1.1 System detection in source files

72

This is perhaps the simplest of all configuration types. If your configuration requirements are static and depend *only* on the operating system being used, then you can put the logic directly in source files. A typical definition in C looks like this:

```
#if defined _WIN32
#include<windows.h>
#elif defined __linux
#include<sys/inotify.h>
#else
#error Unknown platform
#endif
```

This would include the `windows.h` system header on all Windows platforms, a Linux system header on Linux and would fail to compile on all other platforms. This solution works for simple cases but has several downsides. It is not particularly readable or user friendly. You also need experience to know all the platform defines by heart when writing the code.

The second downside is that for many includes this approach gets very complicated. You may need to do branching based on the platform, compiler and even processor used. Decision trees like “if on linux and 32 bit processor or macOS and 64 bit processor but not if the operating system is older than XX” quickly become unreadable. Unfortunately many configuration choices do not neatly follow operating system boundaries, which makes this a real world problem.

The third issue is that it adds to the work needed to port the software to a new platform. Every one of these checks must be updated to deal with the new platform. Porting always takes some work, so this is not such a huge issue, more of an annoyance.

7.1.2 Platform specific files

In this approach configuration dependent code is put into their own files and the build system is told to pick the correct one based on the system. Suppose we have a function called `do_something` which requires a different implementation for each platform. Thus you would have files such as `something-linux.c`, `something-windows.c` and so on and their contents would look like this:

```
#include<current_system_header>
```



```
void do_something() {  
    /* Platform specific implementation here. */  
}
```

73

Choosing the correct file would be done in the build definition:

```
sources = [...]  
if host_machine.system() == 'linux'  
    sources += 'something-linux.c'  
elif host_machine.system() == 'windows'  
    sources += 'something-windows.c'  
else  
    error('Unsupported platform.')  
endif  
  
executable('program', sources, ...)
```

This approach is more readable than the previous one but its other two problems remain.

7.1.3 Compiler flags

The third approach works at a higher conceptual level than the other two. Instead of branching on system and compiler features it instead branches on the desired option. Suppose we have a program that can optionally use an external dependency `foo` if it is available. If not it will use its internal implementation for the same functionality. The C file for this would look like the following:

```
#ifdef USE_FOO  
#include<foo.h>  
  
void do_something() {  
    /* Call into Foo. */  
}  
  
#else  
  
void do_something() {  
    /* Own implementation here. */  
}  
#endif
```

Choosing between these two is done in the build definition based on whether the dependency is available.

74

```
foo_dep = dependency('foo', required: false)
if foo_dep.found()
    use_args = ['-DUSE_F00']
else
    use_args = []
endif

executable(...)
    c_args: use_args)
```

This code will use the `-DUSE_F00` command line switch during compilation when the dependency is found. That causes the `USE_F00` value to be set and the compiler will include and use the external dependency. This example used one file, but the parts could have been split in separate source files that are then conditionally used just like in the previous section.

This approach is fairly nice and usable. Its major downside is that every option adds a compiler flag. This can get unwieldy for projects with many options.

7.2 Configuration files

The established solution for configuration is to write the options in a configuration file, which is also called a *configuration header* in C-like languages. By convention this header has been named `config.h` but this is no longer recommended because development practices have changed. Nowadays it is common to build many projects in one build step (see Chapter 6). Having many headers with the same name may cause the wrong one to be included by accident. A configuration header should therefore always have a unique name such as `<projectname>-config.h`.

Meson provides many different ways of creating the configuration file, so let's start with the simplest one. Suppose we want to create a configuration file `sample-config.h` that has one option that can have value 0 or 1. That is we need to produce a file that looks either like this:

```
#pragma once

#define CONF_OPT 0
```

or like this:

```
#pragma once

#define CONF_OPT 1
```

This requires two things: a *template file* and *configuration data*. Also by convention the template file's name is the same as the resulting file's name followed by a `.in` suffix. In this case the file is called `sample-config.h.in` and has the following contents:

```
#pragma once

#define CONF_OPT @CONF_OPT@
```

In order to generate the output file from the template, we need to tell Meson that it needs to replace `@CONF_OPT@` with either 0 or 1. This is accomplished with the Meson function `configure_file`, which requires three inputs, namely the input file name, the output file name and the values to set. Defining the values is done with the `configuration_data` function. Combining these we get the following piece of code.

```
cdata = configuration_data()
cdata.set('CONF_OPT', 1)

configure_file(input: 'sample-config.h.in',
              output: 'sample-config.h',
              configuration: cdata)
```

You can also pass a dictionary object as configuration data and the behaviour is identical. For the rest of this chapter we will use explicit data objects, though, since they are a bit more convenient to work with.

To convert the template to the final file, Meson will go through it line by line and look for words surrounded on both sides with `@` characters. If that word has been set in the configuration data object, it is substituted in the output file. In this particular case Meson will find `@CONF_OPT@` and replace it with either 0 or 1.

7.3 Advanced configuration options

Meson's text substitution works only on tokens. It does not know anything about the syntax or semantics of the files it processes. Thus it is the user's responsibility to ensure that the output is valid. Perhaps the most common

issue that arises is defining strings in C-like languages, which need to be double quoted:

76

```
#define A_STRING "Some text."
```

You can either put the quote marks directly in the template file:

```
#define A_STRING "@A_STRING@"
```

Or you can put the quotes in the string:

```
cdata.set('A_STRING', '"Some text."')
```

This works but it may look unaesthetic. As this is such a common operation, configuration data objects provide a convenience function to set a value and add double quotes transparently.

```
cdata.set_quoted('A_STRING', 'Some text.')
```

All three of these create the same output. Choosing between them is a question of preference, however the last of these is the recommended one. It is the most explicit and readable of the alternatives.

Configuration in C-like languages is almost always done through preprocessor macros and definitions. It does not have a native concept of **true** and **false**, the values 0 and 1 are used instead. Configuration data has a convenience function also for this case:

```
cdata.set10('TOKEN', boolean_value)
```

This call sets the value of **TOKEN** to 0 or 1 if the argument is **false** or **true**, respectively.

While this simple token based substitution works for most cases, there are cases it can't handle. Some projects require a token to be either set to some value or *not set at all*. There may even be a need to remove any existing definition of a token with the string **#undef TOKEN**. These can not be achieved with pure token replacement, so Meson provides a more flexible configuration method. It can be used by writing a line like this in the template file:

```
#mesondefine TOKEN
```

When Meson encounters this line, it will do a more complex substitution based on the value and type for the given token. The full details are listed in

Table 7.1: How a `#mesondefine NAME` is converted to the output for each supported input value type.

type	sample value	output
string	text	<code>#define NAME text</code>
	"text"	<code>#define NAME "text"</code>
integer	42	<code>#define NAME 42</code>
boolean	true	<code>#define NAME</code>
	false	<code>#undef NAME</code>
undefined		<code>/* undef NAME */</code>

Table 7.1. If we take the example earlier and want to have a token that is either defined to a specific value or not defined at all, we would either set the value to `true` or not define it at all. In the first case the declaration will be transformed to this:

```
#define TOKEN
```

In the other case it would be expanded to this:

```
/* undef TOKEN */
```

which is a comment and thus a no-op.

The final convenience method we shall look into is generating an output file without a template. Adding new configuration options to the build requires modifying two different places: the template file and the configuration data object. Most configuration files only have `#mesondefine` lines in them so it would be convenient to be able to define everything in one place.

This is done by omitting the `input` keyword argument from `configure_file`. This will cause Meson to generate the file automatically based only on values defined in the configuration object. It will iterate over all set values in the object and write them out to file as if there was an input line with a `#mesondefine` for the token in question.

Thus if you have defined the configuration like this:

```
cdata = configuration_data()
cdata.set('DEFINE_ONLY', true)
cdata.set('SOME_NUMBER', 1)
```

78

```
configure_file(output: 'sample-config.h',  
              configuration: cdata)
```

The final generated file would look like this:

```
#pragma once  
  
#define DEFINE_ONLY  
  
#define SOME_NUMBER 1
```

Sometimes you want to provide comments for options. Every `set` method in the configuration data takes a `description` keyword that can be an arbitrary string. It will be written as a comment above the actual value. The output of this declaration:

```
cdata.set('TOKEN', 1,  
         description: 'Some descriptive text here.')
```

would look like this:

```
/* Some descriptive text here. */  
#define TOKEN 1
```

7.4 Introspecting the system

Typical things to write in a configuration file include the program's version number as a string and install paths for looking up resource files on install. They are just pieces of information that the program can use directly. There are usually only a couple of these in each project. More common configuration settings are things that enable or disable some functionality or code paths.

Let's take the example from earlier where we have a project that can use an external third party library or its own implementation. Typically the third party library is more performant, but not available on all platforms. Setting up the configuration for this starts by searching for the dependency and generating a corresponding configuration header:

```
cdata = configuration_data()  
ext_dep = dependency(..., required: false)  
cdata.set10('USE_3RDPARTY_LIB', ext_dep.found())
```

```
configure_file(...)
```

The configuration file template has this declaration:

```
#mesondefine USE_3RDPARTY_LIB
```

When the library is found the generated header will look like this:

```
#define USE_3RDPARTY_LIB 1
```

If the library is not found, the value is 0 instead of 1.

The implementation source file has both the external and internal definitions and choosing between them is done with the `#if` preprocessor define.

```
#include<configuration-header.h>

#if USE_3RDPARTY_LIB

#include<3rdparty.h>

int functionality(int argument) {
    // Implement the functionality by calling the
    // external library.
}

#else

int functionality(int argument) {
    // Implement the functionality with
    // self written code.
}

#endif
```

Storing both implementations in one file is convenient but not mandatory. You can also put the implementations in different files, and only add one of them in the target's list of sources. This is recommended especially if the implementation contains a lot of code. Navigating source files with many `#ifdefs` is not pleasant.

Unfortunately splitting functionality into different files like this is not always possible. Let's take the concept of event driven IO. Linux provides three different functions for that, `select`, `poll` and `epoll`, which are defined in headers `sys/select.h`, `poll.h` and `epoll.h`, respectively. Other platforms provide

some, but not all, of them and they might be defined in headers with different names, such as plain `select.h`.

Trying to write an `#ifdef` declaration that works on all these platforms is difficult and the result is usually unreadable and undebuggable. Worse, it might not even be possible to write such a declaration fully inside the source file. Operating systems add functionality all the time. Simply checking if the operating system used is Linux is not enough, because some versions of Linux might provide the header while older ones do not. On some systems the headers are there and can be used, on other systems they are not.

The only reliable way out of this is not to inspect what platform we are currently running, but instead test *if the header itself is available*. This can not be done from within a source file, because trying to include a non-existing header is a hard error. This test must be done within the build system during the configuration phase.

Meson has native support for compilation checks. They are done with the compiler object that is obtained from the main `meson` object. Taking the `select` example from above, here's how one would check the existence of the two different headers.

```
cc = meson.get_compiler('c')
cdata = configuration_data()
cdata.set10('HAVE_SELECT_H',
            cc.has_header('select.h'))
cdata.set10('HAVE_SYS_SELECT_H',
            cc.has_header('sys/select.h'))
```

The established naming practice for header tests is to start the token with `HAVE_` followed by the header name with slashes and periods replaced by underscores. The tokens are written in all upper case as is common with C preprocessor definitions.

The implementation file can now include the correct header regardless of how it is provided by the system.

```
#include<configuration-header.h>

#if HAVE_SELECT_H
#include<select.h>
#endif
#if HAVE_SYS_SELECT_H
#include<sys/select.h>
#endif
```


This layout is a lot more readable as there are no nested `if` conditions. The compiler object provides many more compilation checks, such as existence of functions and types, sizes of structs and whether a struct provided by the system has an element with the given name in it. A full list of system introspection checks can be found in the compiler object's reference documentation in Section 18.3.

7.5 Printing status messages

As configuration choices are done, one usually wants to print some sort of status messages so the developer can easily see how the system has been set up. Thus far we have used the `message` function to print informal textual messages. This works, but only for simple use cases. Since the output of `message` is printed immediately, it is easy to lose important messages among the rest of them. It also does not provide easy alignment of items over multiple calls.

Since this is a common need, Meson provides a mechanism for printing summary status reports. This is accessed through the function called, unsurprisingly, `summary`. It has a simple data model that consists of *sections* containing multiple *key-value pairs*. The `summary` function can be called at any time and Meson will store the given message internally without printing it. Once configuration has successfully finished, all summary messages are printed.

The behaviour is perhaps easiest to understand with an example. Suppose we have a `meson.build` file that looks like the following.

```
project('DemoProject', 'c', version: '1.0')

summary('mainloop', 'glib')
summary('queue_length', 42)
```

Executing it would produce the following summary output at the end of the configuration run.

```
DemoProject 1.0

    mainloop: glib
    queue_length: 42
```

All summaries are printed under the main project and are pretty-printed. Once the project grows more complicated, you probably want to split the options under different headings. This is done with the `section` keyword argument..

82

```
project('DemoProject', 'c', version: '1.0')

summary('mainloop', 'glib')
summary('length', 42, section: 'queue')
summary('threadsafe', false, section: 'queue')
```

The output of this project looks like the following.

```
DemoProject 1.0

      mainloop: glib

queue
      length: 42
      threadsafe: False
```

The key values to be printed can be strings, integers, booleans or arrays of same. For example a declaration like this:

```
summary('plugins', ['dummy', 'validator', 'performance'])
```

produces the following output.

```
plugins: dummy
        validator
        performance
```

Sometimes it is more convenient to define summary messages to be printed with dictionaries. If you already have one that you use for other purposes (such as defining targets with a `foreach` loop), then printing it with `summary` can be done like this.

```
queue_status = {'length': 42,
                'threadsafe': false}
summary(queue_status, section: 'queue')
```

This leads to the same output message as earlier.

```
queue
      length: 42
      threadsafe: False
```

Like with individual function calls the heading argument is optional. If it is not defined, the dictionary values are put under the project heading.

```
project('DemoProject', 'c', version: '1.0')
toplevel = {'mainloop': 'glib', 'queue_length': 42}
summary(toplevel)
```

83

This produces the same output as the first example.

```
DemoProject 1.0

      mainloop: glib
      queue_length: 42
```

As a final point Meson will keep track of `summary` calls made in subprojects and print them clearly separated. First all subprojects' summaries are printed one by one and finally the master project is printed. Output from different projects is never intermixed.

Chapter 8

Testing

Testing is the foundation on which reliable software is built. Without good tests it is impossible to know if your program behaves correctly. Even more importantly you can't tell it if will keep on working in the future as the code and the world around it change. A good test suite is *fundamental* in increasing developer productivity and by extension development velocity. This may seem counter-intuitive¹, because writing tests does take some time. However the time spent on tests is almost always recovered in decreased debugging time before the project is shipped.

Meson provides a comprehensive set of features and tools to define and run tests. They have been designed to be easy to get started with, but to provide flexibility and power for advanced use cases.

8.1 Defining a test

From a build system perspective the definition of a test is straightforward. A test is an executable whose *return value* defines whether the test succeeded or failed. A value of zero means success and any other value means failure. There are a few exceptions to this rule, which are described in Sections 8.2.4 and 8.2.7 but those are only for special cases.

A simple C program that works as an ersatz test looks like this:

```
#include<stdio.h>

int main(int argc, char **argv) {
    printf("This is a message to stdout.\n");
}
```

¹Especially to managers without first hand development experience.

```

    fprintf(stderr, "This is a message to stderr.\n");
    return 0;
}

```

The program itself only prints some text to both standard out and standard error and then exits with an error code of zero. This means it will always be marked as passing. A real world test program would have more code and return a non-zero value from `main` in case of any test failed.

In Meson tests are defined with the `test` command. A full definition fits in three lines of text:

```

project('simple test', 'c')
exe = executable('simple', 'simple.c')
test('simple', exe)

```

The first line contains the standard `project` definition. The second line creates an executable from a single source file. The third line contains the test declaration, which in its simplest form contains only two arguments: the name of the test and the executable to run.

Tests are run by invoking the `test` target. This runs all tests, collectively known as the *test suite*. The test runner prints status messages to the console as tests are executed.

```

$ ninja test

[0/1] Running all tests.
1/1 simple                                OK          0.00 s

Ok:                                     1
Expected Fail:                         0
Fail:                                  0
Unexpected Pass:                       0
Skipped:                               0
Timeout:                               0

Full log written to <<buildroot>>/meson-logs/testlog.txt

```

The test runner output consists of two parts. First we have one line of text for each test executed with five pieces of information: the test number, the test's name, exit status and the time it took to run the test. These lines are printed as soon as the test has been run. The second piece of information consists of the aggregate status of all tests run. The four different categories are:

Ok The test was run successfully.

Expected fail This test executable has been marked as *should fail*, meaning the test is considered successful if it returns an error, which this test has done. Failing tests are described in Section 8.2.7.

Fail The test returned an error status.

Unexpected pass This is a test that should have failed when run but instead returned successfully. This is considered an error, just like a regular test returning an error code.

Skipped The test reported that it could not be run but that it should not be considered a failure, see Section 8.2.4 for details.

Timeout The test took too long to run, so the test runner mechanism killed it and marked it as a failure, see Section 8.2.5 for details.

The return value of the test runner is the number of tests that failed during the run. This means that the test runner behaves just like any other test: it returns a zero upon failure. This makes it easy to integrate with continuous integration (CI) systems like Jenkins or Buildbot.

When all tests pass, this is usually all the information you need. When things fail, debugging ensues. Most tests write the reason why tests have failed to standard output or error. Meson captures both of these when running the tests and writes them to the file `meson-logs/testlog.txt` in the build directory. The file contains a lot of general data about the test run and also specific data about each individual test.

The general information is fairly noisy, so let's look just at the information specific to the simple test.

```
1/1 simple                                OK          0.00 s

--- command ---
<<buildroot>>/simple
--- stdout ---
This is a message to stdout.
--- stderr ---
This is a message to stderr.
-----
```

As you can see the information is fairly easy to extract. There are three pieces of information that are of interest. The first is *command*, which is the command line used to run the test process. The second is *stdout* containing all

text that the process wrote to standard out, and *stderr*, for all text written to standard error. Using this information it is usually easy to pinpoint the exact reason for a failing test.

8.2 Test properties

8.2.1 Command line arguments

Most unit tests are self-contained, that is, they can be run directly and they require no external input. The rest require either access to some test files or command line arguments to function properly.

A test can be given command line arguments with the **args** keyword argument. Its value is an array that can contain strings, build targets or any mixture thereof. Strings are passed to the command line as is.

```
test('arg_test', some_exe,  
     args : ['--test-mode'])
```

This test would run the given program and give it the one specified command line argument. Build targets are handled specially, they are transparently expanded to paths that point to the target's output file.

```
module_opener = executable(...)  
module = shared_module(...)  
test('module', module_opener,  
     args : [module])
```

This means that the developer does not need to manually build paths to build outputs, the system will handle them automatically.

8.2.2 External test programs

Most test binaries are executables built with the **executable** command, but that is not mandatory. The test executable can also be any executable found with **find_program**. One typical case is to have a custom script that validates that the source code is formatted according to the project's custom style.

```
test_script = find_program('format_validator')  
test('format_check', test_script,  
     args : [meson.source_root()])
```


This snippet finds the validator script (most likely from the current source directory). Then it defines a test that passes the top of the source tree as a command line argument to the script. This is all the script needs to do its task.

The test program can also be an external program. There are many third party test and validator tools available. They are used in the same way as above, except that the build definition should be written so it can be run even on machines where the external tools are not available.

```
third_exe = find_program('external_test_program',
                        required : false)
if third_exe.found()
    test('thirdparty test', third_exe)
endif
```

8.2.3 Environment variables

Some tests require setting environment variables to specific values. This is more difficult than setting command line arguments, since sometimes simply overriding a value does not work. Instead what you need to do is to add new data to an existing variable. A common example is the `PATH` environment variable. Setting it to a specific value makes all system executables inaccessible. Sometime you may want this, but most of the time you want to append new path entries after the existing ones.

The main tool for defining environment variable setups is the `environment` function. It is similar to the `configuration` object as presented in Section 7.2 and it is used in a similar way. First you create an object, then you set values in it one by one and finally use it as a keyword argument in a test definition.

Suppose you have a test that requires the following environment variable settings:

1. Set the value of `TMPDIR` to current build directory.
2. Append the value `/usr/local/bin` to `PATH`.
3. Prepend the current source directory to `XDG_DATA_DIRS`.

This is be achieved with the following piece of code:

```
env = environment()
env.set('TMPDIR', meson.current_build_dir())
env.append('PATH', '/usr/local/bin')
env.prepend('XDG_DATA_DIRS', meson.current_source_dir())
test('envvartest', test_binary,
```

```
env : env)
```

90

All methods on the environment variable object take an optional **separator** keyword argument. It specifies the character that will be used to join the strings. It defaults to the platform's default separator character which is `;` on Windows and `:` on all other operating systems. A detailed description on how the different methods are interpreted w.r.t. existing environment variables can be found in Table 18.1 on page 214 in the reference section.

8.2.4 Skipping tests

Usually when a test can not be run, a reasonable solution is to not define it at all. The most common case is a test that run only on one operating system. It makes little sense to try to run (or even compile) Linux specific tests on Windows so leaving them out is the right thing to do. Sometimes things are trickier and you can only detect that a test can not be run when executing the test.

Typically these problem cases have something to do with the runtime properties of the current operating system or hardware. Suppose for example that you want to create an executable that uses advanced SIMD instructions, such as AVX, if they are available but gracefully falls back to a software implementation on processors that do not support the given extensions. It is possible to compile source code with AVX support on any processor, but if the compilation machine's CPU is too old, the AVX code path can not be tested. Marking the test as passed in these cases would be misleading, since the test has not actually been run.

Instead of passing, the test should be marked as skipped. This makes the result clearly visible in aggregate statistics. Tests are marked as skipped if they exit with a return code of 77. This may seem strange and arbitrary, but it is an old established Unix tradition², which Meson follows to maintain compatibility with existing projects.

8.2.5 Timeouts

Each test is given a certain amount of time to run, by default 30 seconds. If a test takes longer than that to for whatever reason, the test suite runner will unconditionally kill the test process and mark the test as failed.

30 seconds was chosen as the default value as a compromise. It is long enough that almost all tests should have enough time to pass. If your tests hit this limit, they are probably too big and/or doing too much so you might consider splitting them into separate parts. Sometimes there are cases when this can not be done, such as tests working on large data sets.

²In fact for many people *strange and arbitrary* and *Unix tradition* are synonymous.

In these cases you need to increase the timeout value, which is done with the `timeout` keyword argument.

```
test('slow_test', exe, timeout : 60)
```

91

This slow test would have 60 seconds to run before it is killed by the system.

Sometimes you may need to alter the timeout value of all tests rather than just one. This can be done by invoking the test runner with custom arguments which will be described in Section 8.3.5.

8.2.6 Parallel tests

Meson differs from many classical test runner environments by being parallel *by default*. It uses all the compute power on the test machine automatically. What this means in practice is that if your machine has n processors, Meson will run n test executables in parallel. Most of the time the only thing visible to the end user is that their test runs finish faster.

There are cases where automatic parallelisation is harmful. The first consists of tests that use multiple threads or processes internally. These tests typically scale themselves to use all processors on the machine. If you have several of these running at the same time they will end up fighting for resources, which causes slowdowns and eventually killing of all test processes due to timeouts.

The second group of tests require exclusive access to some unique resource. Typical cases include running a server on a socket with a specific port number, owning a specific service name on DBus or having exclusive access to a specific file on the file system. Running two of these tests at the same time would cause test failures. Either one of the tests fails immediately because it can not get the unique resource, or both of the tests *think* that they own the resource but don't. During tests the inputs and outputs of these tests can get randomly intermixed leading to very confusing debug sessions.

Meson's way of avoiding these issues is that you can mark a test as *nonparallel*.

```
test('noparallel', exe, parallel : false)
```

When this test is being executed, Meson guarantees that no other test process is running at the same time. The downside is that this slows down the test suite since only one test is being run. Because of this, tests should be marked as non-parallel only when strictly necessary. Often it is possible to refactor the code to be more testable so it can be run in isolation and thus in parallel with other tests.

8.2.7 Failing tests

92

Sometimes failure is not only an option, but the *expected outcome*. Suppose you are writing a file format parser and want to verify that it does not accept malformed files. In these cases the parser program should exit with an error. If the test program were to return successfully it would mean incorrect behaviour.

Test programs that are expected to fail can be declared with the `should_fail` keyword argument.

```
test('failing_test', executable,  
     should_fail: true)
```

In this case the test is marked as passed if the program's exit code is not zero. This check only applies to the pass/fail status of the test. This presents an additional problem, because test executables can be run both as regular tests and as failing tests. A test executable does not know how it is being run, but there are cases where it would need to report failure regardless of how it is being used. If the program detects that its internal data structures have become corrupted, it should be able to signal a fatal failure. This can be done by exiting with the return code 99. It is always interpreted as a test failure regardless of whether the test has been marked as should fail or not.

If the program has an exit code of 77 it will be marked as skipped and if it takes longer to execute than the timeout value, it is marked as failed due to a timeout.

8.2.8 Changing the working directory

By default Meson makes no guarantees about where the program is being run from. Different backends have different layouts. If you need to ensure that a specific test is run from a specific directory, you can specify this with the `workdir` keyword argument.

```
test('workdirtest', executable,  
     workdir: meson.current_source_dir())
```

This keyword argument should be needed only rarely, usually by tests that expect to be run at the root of a directory tree containing test data. Most tests should be written so that they can be run from any directory, that makes them easier to run manually when debugging. Those that do not can be refactored to take the path as a command line argument.

8.3 Advanced testing using the test tool

All testing thus far in this chapter has been done with the default `test` target that can be invoked from the backend. This is simple and straightforward but also fairly limiting. Meson ships with a testing tool that has many options and features for advanced testing scenarios.

The test program can be run with `meson test`. Running this command in the build directory is equivalent to running `ninja test`. It will make sure that the build directory is up to date (equivalent to `ninja all`) and then run the default set of tests.

8.3.1 Skipping the rebuild step

One of the main goals of Meson is reliability. This is why it always ensures that the code is always built and up to date. There is nothing quite as frustrating as debugging code for a long time only to realise that the executable you are running is stale. Building everything can take a lot of time, though, so sometimes you may explicitly tell Meson not to rebuild the code, but just run the tests. This is achieved with the `--no-rebuild` command line argument.

```
$ meson test --no-rebuild
1/1 simple                                OK          0.00 s

Ok:                                     1
Expected Fail:                         0
Fail:                                  0
Unexpected Pass:                       0
Skipped:                               0
Timeout:                              0

Full log written to <<builddir>>/meson-logs/testlog.txt
```

8.3.2 Running individual tests

The simplest, and possibly the most commonly used, feature is to run only one or few tests rather than all of them. This can be achieved by listing the test names you want to run as command line arguments. Suppose you have a project that has ten tests named `test0`, `test1`, ..., `test9`. If you wanted to run only tests 3 and 7, you'd run Meson test like this:

```
$ meson test test3 test7
```

94

8.3.3 Utilising test suites

As the number of tests in a project grows, it often becomes too slow to run every test on every change. For example you might have a subset of tests that deal with data parsing, a different (but not disjoint) set for database communications. There may even be a bunch of tests that are slow to run and you'd want to execute all tests *except* those.

All of these use cases can be handled by arranging tests into *suites*. This can be achieved by specifying one or more free form text labels using the `suite` keyword:

```
test('parse_and_db', exe,  
     suite : ['parse', 'database'])
```

This test has been tagged to be part of the `parse` and `database` test suite. By default tags do nothing, they are only looked at when the `--suite` argument is used. The following command runs only tests that are part of the `parse` suite:

```
$ meson test --suite parse
```

You can also specify multiple suites.

```
$ meson test --suite parse --suite database
```

This would run tests that are in suite `parse` or `database` or both.

There is also a corresponding `--no-suite` that runs all tests *except* those that are part of the specified suite.

```
$ meson test --no-suite slow
```

These two switches can be used at the same time and their effects are cumulative. To execute all tests in the `parse` suite except those that are also tagged as `slow` you would use the following command.

```
$ meson test --suite parse --no-suite slow
```

8.3.4 Test executable wrapper

There are many third party tools that can be used for improving and enhancing the results of tests. Perhaps the most known of these is the Valgrind [16] family of tools. They work by running an executable in a virtualisation environment looking for common programming errors, such as buffer overflows.

Valgrind and other such tools are simple to use. Rather than running `testcommand -arg`, you run `valgrind testcommand -arg`. That is, adding the inspection tool's binary to the test command is all that is needed. Meson's test tool has the `--exe-wrapper` command line argument for this.

```
$ meson test --exe_wrapper=valgrind
```

This command runs all tests but invokes them via Valgrind. The wrapper argument is split according to regular shell rules, so if you need to pass command line arguments to the Valgrind command invocation, you have to put them in the same argument. Many programs have spurious warnings when run under Valgrind that can be ignored with a *suppression file* specified as a command line argument to Valgrind.

```
$ meson test --exe_wrapper='valgrind --suppressions=supp_file'
```

8.3.5 Timeout multiplier

The big downside of wrapper tools is that they slow down the tests. A typical factor is 10×, but slowdowns up to 1000× are not unheard of. That causes every test to fail due to a timeout. Since manually editing the build definitions to increase the timeout values would be impractical, Meson allows you to temporarily increase the timeout value on all tests.

```
$ meson --timeout-multiplier=100
```

In this case all timeouts are 100 times as long as specified in the file. That is, if a test has the timeout value of 5 seconds, with this flag it would be 500 seconds, or just over 8 minutes.

8.3.6 Other arguments

In addition to the major features discussed above, `meson test` has a bunch of minor features and options that do not warrant a section each, but which are very useful.

The first is the flag to control the maximum number of processes running at the same time. This value defaults to the number of CPUs on the machine but can be set explicitly with an argument like `--num-processes=4`.

Some test failures are *sporadic*, that is, they only happen every now and then. If you fix one of these bugs it is difficult to prove that problem has truly been eliminated. The only reasonable approach is to run the tests sufficiently many times. This can be achieved with the `--repeat` command line argument that takes a number, such as 1000, and runs the test suite as many times a specified.

The final argument is `--gdb` which runs each test under the GNU debugger [17]. If the test crashes, the developer is dropped to GDB's command shell and they can debug the crashed program. Because debugging sessions can take an arbitrary amount of time, timeouts are disabled while using GDB mode. This argument becomes even more useful when combined with the previous one like this:

```
$ meson test --repeat=10000 --gdb sporadic_fail_test
```

This invocation would run the failing test 10 000 times under gdb. Once invoked the test can be left running in the background, possibly overnight. The full run will either pass or end up with a debug session at the exact point of the crash.

8.4 Defining custom test setups

The command line tool is useful for experimenting with testing options. Once you find a combination of settings that works for your project you may want to store them for later usage. An age-old Unix solution is to write a shell script with your commands and store that in your project. Meson provides a simpler way of storing this information directly in your build definition file as a *test setup*.

```
supp_file = meson.current_source_dir() / 'my_suppressions.txt'
add_test_setup('valgrind',
    exe_wrapper: ['valgrind', '--suppressions=' + supp_file],
    timeout_multiplier : 1000)
```

This defines a setup named `valgrind` that runs all tests under Valgrind. The keyword arguments are the same as the command line arguments to the tester program (with the exception that commands are arrays in Meson but strings in the shell). The testing system does not guarantee which directory the tests are run in, so we build an absolute path to a suppression file in the current source directory. This setup would be used like this:


```
$ meson test --setup=valgrind
```

In addition you can also specify environment variables to set as well as the use of gdb.

```
e = environment()
e.set('EXTRA_VALIDATION', '1')
add_test_setup('strict_gdb',
    env: e,
    gdb: true)
```

97

8.5 Benchmarks

All tests discussed thus far have been about verifying the correctness of the problem. Either all tests pass, and the program is presumed correct or one or more of them fail proving the program broken. This is only one slice of the testing pie. A different, but just as important, part is performance.

From Meson's point of view, benchmarks and regular tests are almost identical. They are defined and run in a similar way. The major difference is that Meson will not run benchmarks in parallel. Defining benchmarks is simple:

```
benchmark('speedcheck', exe)
```

The `benchmark` command takes all the same keyword arguments as `test`.

```
$ ninja benchmark
1/1 speedcheck                                OK          0.00 s

OK:      1
FAIL:    0
SKIP:    0
TIMEOUT: 0

Full log written to <<builddir>>/meson-logs/benchmarklog.txt
```

Running benchmarks with the test tool is equally simple.

```
$ meson test --benchmark
```

All command line arguments that were used with tests can also be used with benchmarks and they behave in the same way.

Chapter 9

Installing

Getting a project configured, built and tested is a good start. However a program that only runs on your own computer is not as useful as one that can be given out to other people to run on their machines. The process of taking a built program and all its data files, separating it fully from the source code and turning it into a self contained application is called installation. The end result is a package or bundle of some type that can be deployed on other machines. Confusingly the task of deploying a bundle is also called installing. In this chapter we will only deal with the first type of installation.

9.1 Directory layout

Perhaps the biggest individual task of installation is one of file layout. Source code trees, and thus by extension build tree layouts are usually optimised for development convenience. The file layout of applications is conversely optimised for running and deployment. Installation takes files in one layout and converts them to the other.

Every platform has its own recommended (or mandated) file layout. The most common are the classical Unix layout, the macOS application bundle layout and Windows application layout.

9.1.1 The Unix layout

The file layout used on most Unix systems today comes pretty much unchanged from the original Unix file system hierarchy defined in the 1970s. It is based on a hierarchical directory tree with specific directories. Ignoring certain low level system programs, everything gets installed under the `/usr` directory. This is called the *installation prefix*. Everything else in the installation is defined

100

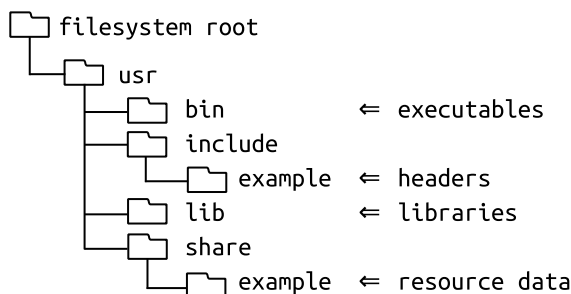


Figure 9.1: The most common install directories required on a Unix platform by an application called *example*.

relative to this prefix. The advantage of this is that it becomes easy to install the program in a different location simply by changing the prefix. A typical alternative prefix is `/usr/local`. On Linux systems this directory is seldom used, but on the other hand on systems such as FreeBSD everything not part of the core system is installed under `/usr/local`. A well behaved program should work regardless of the installation prefix. For the rest of this chapter, we are going to assume that prefix has been set to `/usr` unless otherwise specified.

Within the installation prefix are many directories each of which is meant for specific types of files. The most common directories can be seen in Figure 9.1. The only directory that applications own completely and can use for any type of file they want is the resource directory under `share`.

Unix installations are static and read only. That is, once files have been written in the directory tree they can not be changed any more. Data that changes at runtime needs to be stored elsewhere, usually in the user's home directory. Most unix systems, especially Linux, do not install files in `/usr` manually but with a dedicated piece of software called a *system package manager*. This system takes care of installing and uninstalling packages safely, as changing the root file system of a running computer is a risky proposition. Package managers have other tasks as well, such as ensuring that all dependencies of a program are available before installing the app. Some of the most known system package managers include *apt*, *dnf* and *zypper*.

Installing everything in the system root directories has both advantages and disadvantages. Perhaps the biggest downside is that an application built on one Linux distribution can usually not be run on any other distro or even a different version of the same distro. A recent trend has been to move applications away from distro packages and to instead provide self contained application bundles that can be run on all conforming platforms. Most of them work by using

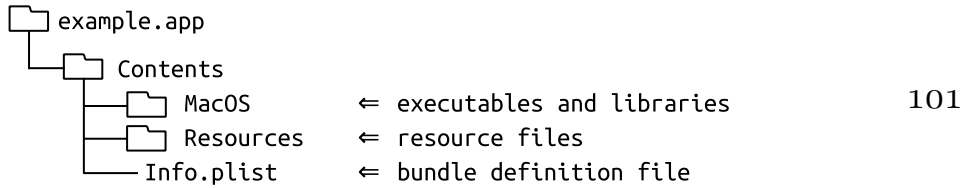


Figure 9.2: A directory layout for a macOS application bundle.

operating system kernel functionality to create a containerised file system that can be run in isolation. The most adopted solution at the time of writing is Flatpak [12]. Any project that supports installing to an arbitrary prefix will automatically work with these systems.

9.1.2 The macOS app bundle layout

Apple’s macOS is a Unix system at its core but it behaves *very* differently from other unixes currently in use. Users are very strongly advised not to install anything to the system directories. Instead all applications should be provided as self contained *application bundles*. A bundle is really nothing more than a directory tree of files in a specific format. The operating system will detect bundles and show them to the user as an application that can run. The basic file layout of an app bundle can be seen in Figure 9.2.

There are two things here that make the operating system treat this directory tree as an application. The top level directory name must have the suffix **.app** and a file called **Info.plist** must be in the **Contents** subdirectory. If both of these conditions hold true and the file is in the required XML format the directory works as an application. You can, for example, start it by double clicking on it. This setup is typical for application bundle formats in general. Most of them consist of an archive of files and a magic file identifying the archive as an application bundle.

The sample project places all binaries in the **Contents/MacOS** directory. For executables this is mandatory but for libraries it is done merely for convenience. More complex projects would need to set up frameworks and other constructs in the specific format and layout specified by the operating system.

The **Resources** directory holds data files and corresponds to **share/example** directory on Unix. Certain files must be in this directory according to very specific rules. Application private data, such as textures in games, can be placed in the directory in an arbitrary structure. The exact rules depend on the platform so you should always refer to documentation provided by the operating system

Table 9.1: Install locations for executable `example` if prefix is `/usr` and `bindir` is `bin`.

type	value	install location
not defined		<code>/usr/bin/example</code>
relative path	<code>sbin</code>	<code>/usr/sbin/example</code>
absolute path	<code>/sbin</code>	<code>/sbin/example</code>

vendor for specific layout rules in bundles.

9.1.3 The Windows application layout

The classical Windows application model could affectionately be called the *bag of files* model. There are no specific requirements, so a common approach is to put all files in a single directory with possibly a subdirectory or two for assets.

The only real requirement that Windows has is that executables and shared libraries, called DLLs, must be in the same directory in order to be runnable. They can, in theory, be put in separate directories but that complicates things for no real benefit.

9.2 Installing build targets

By default Meson does not install anything. Build targets can be installed with the `install` keyword.

```
executable(..., install: true)
```

This will install the target in the system default directory for the given file type. Specifically executables will be installed to the directory pointed to by the option `bindir` and libraries are installed to path pointed to by the option `libdir`. Every function that generates output has an `install` keyword argument that can be used to install the result. Thus it is easy to see at a glance whether some target is installed or not.

These other target types may additionally require the `install_dir` keyword argument specifying the install destination. The same argument can also be given to build targets to install them in a non-standard directory. The final install locations corresponding to different types of values can be seen in Table 9.1.

The path behaviour is the same on all install targets. If the install directory is undefined, then the system default is used. Relative paths are interpreted to be relative to `prefix` and absolute paths are used as is.

9.3 Installing other files

103

In addition to build targets one often needs to install resource files as well. A computer game might have textures, 3D models and music, whereas an office suite might have template documents. Meson provides two functions for installing files, `install_data` and `install_subdir`. In addition there is a helper function `install_man` for installing man pages, but it will not be discussed in detail. Those interested should look up reference documentation in Section 19.30.

The simpler of the remaining two functions is `install_data`. You give it a list of files and the path to install them into.

```
# Both files go to /usr/share/mygame
install_data('skin.png', 'wizard.mdl',
             install_dir: 'share/mygame')
```

Note that path segments in input file names are dropped on install.

```
# Even now both files go to /usr/share/mygame
install_data('graphics/skin.png',
             'models/wizard.mdl',
             install_dir: 'share/mygame')
```

Writing out install commands by hand gets tedious once you have more than a couple of files to install. The common solution to this is to set up project file layout so that resource files are in the same layout as they will be in the final install location. Then you can install the entire subtree in one go using `install_subdir`.

```
# Assuming a file tree like this in your project tree:
#
# resources
#   graphics
#     skin.png
#   models
#     wizard.mdl
#   sounds
#     casting.mp3

# The output of this:
install_subdir('resources', install_dir: 'share/mygame')
```

```
# Will end up like this
#
# usr
#   share
#     mygame
#       resources
#         graphics
#           skin.png
#         models
#           wizard.mdl
#         sounds
#           casting.mp3
```

A common use case is that you need to install the subdirectory but without the top level `resources` segment. That is, so that the `graphics` directory is directly in the `mygame` directory. This can be achieved with the `strip_directory` keyword argument.

```
install_subdir('resources',
               strip_directory: true,
               install_dir: 'share/mygame')

# Will look like this:
#
# usr
#   share
#     mygame
#       graphics
#         skin.png
#       models
#         wizard.mdl
#       sounds
#         casting.mp3
```

9.4 Running the install

Install is initiated by invoking the `install` target. With the Ninja backend this is done by calling `ninja install`.


```
$ ninja install  
Installing libstat.a to /usr/lib  
Installing prog to /usr/bin
```

Meson sets up proper dependencies so that the build is always up to date before the install step is run. If the code can not be built, Meson will exit with an error and no install steps are run. This ensures that you don't install stale or broken code by accident.

The default target does the right thing most of the time. Like with testing, Meson provides a richer install command that can be invoked with **meson install**. By default it does the same thing as basic install but it also accepts two command line arguments. The first is **--no-rebuild** which merely runs the install without verifying that everything is up to date. If you use this argument it is your responsibility to ensure that everything is up to date.

The second argument is **--only-changed**. By default Meson will copy every file to be installed. If there is already a file with the same name it will be replaced unconditionally. If the project has lots of large data files then incremental install commands are as slow as the first one. This command line argument changes the install behaviour so that files will only be overwritten if their file system timestamps are older than the source file being copied. In practice this means that consecutive install commands will only copy over files that have changed since the previous install.

9.4.1 Staging installs with DESTDIR

Install places files in their final runtime locations, such as **/usr/bin** for executables. This is problematic because on most platforms you are not permitted to write files in system directories. Files may only be installed in those directories using packages. Thus you need to first install the files in some staging directory, convert the contents of that directory into a package and then install the result.

A straightforward solution would be to set up the prefix so that it points to the staging directory rather than the actual system. A potential value of prefix using this scheme would be **/tmp/staging/usr**. This is not recommended because it fails quite badly when resource files enter the picture.

An established practice in Unix software development is to store the install prefix and possibly the project's data directories inside the executable itself as discussed in Chapter 7. This would cause the programs to look up files from the staging directory (**/tmp/staging/usr/share/program**) rather than the actual system directory (**/usr/share/program**) at runtime meaning none of the resource files would be found.

Fixing this issue requires some mechanism where projects are configured

Table 9.2: Environment variables set when running install scripts.

106

name	type	sample value
MESON_SOURCE_ROOT	path to source root	/path/to/source
MESON_BUILD_ROOT	path to build root	/path/to/build
MESON_INSTALL_PREFIX	installation prefix	/usr
MESON_INSTALL_DESTDIR_PREFIX	full install location	/tmp/staging/usr
MESON_INTROSPECTION	introspect command	meson introspect

with their final runtime prefix but get installed into a completely different staging directory while preserving the same directory tree layout. The established standard way of doing this is the `DESTDIR` environment variable, which all install tools are expected to obey. The variable defines an absolute path segment that should be prepended to the output name of all installed files.

The environment variable can be set in the standard Unix shell way.

```
$ DESTDIR=/tmp/staging ninja install
[0/1] Installing files.
Installing libstat.a to /tmp/staging/usr/lib
Installing prog to /tmp/staging/usr/bin
```

9.5 Custom install tasks

There is more to installation than copying files. Since these steps are usually highly project-specific Meson does not attempt to provide functionality for all possible things. Instead it provides a way to run custom installation scripts after basic install steps have finished. An install script can be added using the `meson` object:

```
meson.add_install_script(command, arg1, arg2)
```

Like with all Meson functions that take a command to be run, it can be an executable, custom target or the result of `find_program`. It can also be a plain string, in which case it is internally converted to a program by using the `find_program` command. The remaining args are passed as command line arguments to the script. Meson will also set environment variables listed in Table 9.2 when the command is run.

Most entries in the table are fairly self-explanatory but a few might require some discussion. The variable `MESON_INSTALL_DESTDIR_PREFIX` contains the install prefix with `DESTDIR` prepended. This is the directory where the script should put all its files. It is recommended to use this variable rather than trying to create it manually, because combining `DESTDIR` with the prefix has some edge cases.

The second unusual variable is `MESON_INTROSPECT`. It merely provides a way to run Meson introspection commands¹ from the install script. The actual command can vary a lot between operating systems. Here are a few possible values:

environment	value
default	<code>/usr/bin/meson introspect</code>
custom install	<code>/home/username/meson/meson.py introspect</code>
custom Python	<code>/usr/bin/python34 /usr/bin/meson.py introspect</code>
Windows	<code>C:\Program\ Files\meson\meson.exe introspect</code>

Using the environment variable means that install scripts don't have to implement lookup heuristics for Meson's executables, but instead they will always work. Note that the string usually contains multiple parts, the command and its argument. If your script is implemented in a language that requires commands to be arrays rather than shell command strings, you need to split the string yourself using the tooling provided by the language. Simply splitting by spaces is not reliable, it will break if your paths have any spaces in them.

9.6 Other things that happen during install

Even though the user-visible part of installation looks like file copying, behind the scenes things are different. Merely copying files from the build directory to the install location is not enough, and doing so would lead to broken installs. Meson does additional work required by the system and changes the installed files. These changes deal with executables and shared libraries.

Providing shared libraries from system directories requires following a set of fairly complex guidelines. For example on Linux and some other unixes the output file name of a library `example` is *not* `libexample.so` as one would expect. Instead there should be a symbolic link named `libexample.so` that points to the actual file with very specific naming conventions based on its name and version. A detailed overview of the problem is out of scope of this book, suffice it to say that Meson takes care of all this boilerplate behind the scenes. Interested

¹At the time of writing the introspection functionality is still in flux, so it is not discussed in this book

readers are advised to look up the reference documentation for their platform to find out more.

108

Another important change has to do with a concept called *rpath*. When you run an executable that uses shared libraries, the system's dynamic linker will only allow those files to be found from certain directories. This is a basic security feature to prevent malicious programs from hijacking system libraries.

The downside of this is that if you build a shared library and an executable using it, the executable can not be run directly from the build directory. The executable file formats used on most platforms (ELF on most Unix systems and Mach-O on macOS) allow you to write additional information inside the output files. These *rpath* entries list additional directories where shared libraries can be loaded.

Meson automatically sets up *rpath* entries so that executables can be run directly from the build directories. It will automatically remove these entries from the files it installs. Spurious *rpath* entries are forbidden in system libraries because they can cause security issues as well as weird behaviour and difficult bugs.

Windows does not support *rpath* entries at all. Meson tries to emulate the behaviour as best it can, but in the general case programs can not be run directly from the build dir on Windows platforms.

9.7 Accessing data files before and after install

Having an application access its own data files turns out to be a fairly tricky problem. A typical cross platform application should work with at least the following ways:

1. When run from the build directory, it should access data files in the source directory.
2. When installed on the system, it should access data files from the system install.
3. When run as a relocatable app bundle, it should access data files in its own bundle, that is, relative to the current executable.
4. In all cases it should provide a way to override the location of data files with e.g. an environment variable.

There are many almost-working solutions to this problem. For example the first item can be worked around by requiring that the program is always installed and run from the install location rather than from the build dir. This works but

is tedious, slow and it's surprisingly easy to forget to run the install command before starting the program.

Unfortunately there is no ready made solution that works on all these different cases. This is something each project needs to do on their own based on their specific requirements. A typical solution would be to have a dedicated function, say `open_data_file` that is the only way to access data files. Its implementation might look like this (in C-like pseudocode):

```
FILE* open_data_file(const char *file_name) {
    if(envvar_set("PROG_DATADIR"))
        return fopen(envvar_value("PROG_DATADIR") + file_name);
    if(is_run_from_build_dir()) {
        return fopen(path_to_source_dir + file_name);
    }
    return open_file_installed(file_name);
}
```

That is, if file location has been overridden with an environment variable, use that. If we know, somehow, that the program has been invoked from the build dir, load files from the source dir. Otherwise call into a platform specific file open method. A pseudocode solution to the platform specific part might look something like the following.

```
#ifdef __APPLE__
FILE* open_file_installed(file_name) {
    // Open file from within the current bundle.
}
#elif _WIN32
FILE* open_file_installed(file_name) {
    // Open file according to Windows rules.
}
#else
FILE* open_file_installed(file_name) {
    return fopen(INSTALL_PREFIX + file_name);
}
#endif
```

This code only works if the layout of resource files is the same in the source directory as it is in the final installed directory. This is the recommended way to lay out resources. It also allows you to install the entire resource tree with one `install_subdir` command. There are many projects that do not follow this rule, but instead keep, for example, in separate directories in source root. Typically all graphics are in a directory `graphics`, sound files in `audio` and so on.

Both of these approaches work and can be used. Neither of them is “the right way to do things”, they just have different tradeoffs. Having the same layout in the source tree and install tree is a bit more work when resource files are created and manipulated, as they are slightly deeper in a file system hierarchy. Top level directories are easier to manage, but they require more code to install. They also require more code to access when run from the build dir, as all file paths are different.

9.7.1 Packing resources to an archive

As a final twist to installation, sometimes resources are not provided individually but as a potentially compressed single resource image. This is most common in computer games, where resource packing is used to, for example, minimise image size by using custom compression schemes and trying to make the game harder to reverse engineer and exploit.

Generating a compressed image is usually slow and computationally expensive, thus it should not be used during development, only when creating the final result. This typically happens during integration tests that run daily and when shipping the final product. This requires updating the resource loading code to work in both cases transparently.

The build system changes are straightforward. The difficult part is writing the functionality for creating the archive. This is always project dependent and it is best written as a standalone script that hides the implementation. Then we need a project option to decide whether to install resources directly or to pack them. Putting these two together we get the following build definition snippet.

```
if get_option('pack_resources')
    meson.add_install_script('resource_packer.py',
                           meson.current_source_dir())
else
    install_subdir(...)
endif
```

Chapter 10

Project options

All projects start simple. There is one executable. It does one thing. Everything is fine. Then it needs to do two things. Then three. Four. Five. Eventually you find a need for some optional feature, such as enabling extra debug logging. When faced with this problem, many people first solve it by editing their build definitions so that the functionality can be removed by commenting out some declaration.

As the project size grows the number of things that need toggling usually grows. After about three such options it becomes inconvenient to configure the project by editing the build definition. What is required is some kind of mechanism to change the build configuration from the outside without editing the build definitions. In Meson this is achieved by *build options*.

10.1 Builtin options

Meson provides several options automatically for every project. These are for global settings that most projects require. The most common of these are the various *install paths* that specify where files go when installed as discussed in Chapter 9.

The remaining builtin options are used for specifying elementary properties of the build such as optimisation levels, debug info, compiler warnings, language standards and many more. Even though there are many options, in day to day development most people will only need to care about a handful of them. The most common of these being **buildtype**, which can be used to quickly toggle between debug and optimised builds. Obtaining the full set of options will be explained in Section 10.4

10.2 Declaring and using project options

112

Each project can specify an arbitrary number of options. They are defined in a file called `meson_options.txt` located at the project source root. An option for toggling extra debug code can be written like this:

```
option('extradebug', type: 'boolean', value: false,
       description: 'Enable additional debug code.')
```

By itself an option does nothing, it's just a variable. Using it requires adding functionality in the build definition.

```
project('simpleoption', 'c')
if get_option('extradebug')
    add_project_arguments('-DEXTRA_DEBUG', language: 'c')
endif
```

If the option is set then all C compilations get the extra arguments.

10.3 Defining options

Options are defined with the `option` function that can only be called from the option file. Its first argument is the name of the option. The function accepts many keyword arguments, some of which are common and some which can only be used with certain other option types. The common keyword arguments are:

type defines the type of the variable and can be one of `string`, `boolean`, `integer`, `combo`, `array` or `feature`.

description is a freeform string comment.

value is the default value for this option.

10.3.1 String option

A string option is a freeform UTF-8 string. Its definition takes no additional keyword arguments and its default value is the empty string.

10.3.2 Boolean option

A boolean option can only hold the values `true` or `false`. Its definition takes no additional keyword arguments and its default value is `true`.

10.3.3 Combo option

A combo option can take one value of a predefined set of strings. The list of possible values is set with the `choices` keyword argument whose value is an array of strings. If the default value is not defined, the first value from the array is used.

113

```
# The value can only be one of the four main headings.
option('heading',
      choices: ['north', 'east', 'south', 'west'])
```

10.3.4 Integer option

An integer option is a value. Optionally you can set the maximum and minimum value with the `min` and `max` keyword arguments.

10.3.5 Array option

An array consists of one or more strings. By default there are no limitations on the contents. If the `choices` keyword argument is defined then only strings that are in the choice array may be put in the value.

```
# This option can have values like
# ['red'] or ['red', 'blue'] or []
# but not
# ['yellow']
option('colors', type: 'array',
      choices: ['red', 'green', 'blue'])
```

10.3.6 Feature option

A feature option is a special kind of combo option. It can have one of three values: `enabled`, `disabled` or `auto`. When `get_option` is called on a feature object it does not return a string, but a special feature object. The only thing you can do with one is to pass it to the `dependency`, `find_program` and `add_languages` functions and the compiler object's `find_library` method as the `required` keyword argument.

The option overrides Meson's lookup functionality. The `enabled` value is the same as specifying `required: true`. The value of `disabled` immediately returns a not found object without actually looking. The value of `auto` is special as it ties the value to a separate builtin option called `auto_features`. In a weird inception-y twist that option *also* has possible values of `enabled`, `disabled` and

`auto`. In this case the `auto` value is identical to specifying `required: false` as the keyword argument.

114

This setup may seem weird and convoluted, and for single options it is. The design starts to make more sense when there are many optional components, such as plugins¹. Suppose you have a project with a dozen plugins, and each needs to be enabled or disabled individually. This implies that every plugin needs to have its own option. However toggling every single option is tedious so you'd also want to have a way of switching all at once. This can be achieved by setting each individual toggle option to `auto`.

Now you can switch all plugins on or off with the global `auto_features` option. Overriding the setting for any individual option can still be done by switching that option from `auto` to, say, `disabled`.

10.4 Exploring and setting option values

One thing about Meson options that often confuses new users is that you can *never set option values from within build files*. They are always set from the outside. Build files can request certain default values to be used when the project is being configured from the first time but they are not guaranteed to be used. Once an option has any value, it can never be changed from within the build file, even during reconfiguration. The value it has when reconfiguration begins will remain until it has fully finished.

Requesting new default values is done in the `project` method call.

```
project(..., default_options: ['optname=value'])
```

This snippet tells Meson that when the project is being configured for the first time, the value of option `optname` should be `value`. This statement is *only* considered on the first run. On subsequent configurations it is ignored *even if you change this value in the project call*, because the option already exists and has a value. If the user has specified a value on the command line with `-Doptname=value2`, it will override the default value set in `project`.

The way to set options is to use the `meson configure` command line tool. The default way of using it is to execute the `configure` command in an existing build dir. This will list all global and project options and their current and possible values.

The output of the program is too wide to fully fit on this page, so here is a highly condensed and edited version of the output. The real output for each option has e.g. a brief description about the option and what it is used for.

¹This feature was originally designed for the GStreamer multimedia framework, which has 100+ plugins

```
$ meson configure
Core properties:
  Source dir <<source dir>>
  Build dir  <<build dir>>

Core options:
  Option                Current Value Possible Values
  -----
  buildtype             debug                [plain, debug, ...]
  debug                 true                [true, false]
  optimization           0                  [0, g, 1, 2, 3, s]
  warning_level          1                  [0, 1, 2, 3]

Base options:
  Option                Current Value Possible Values
  -----
  b_lto                 false                [true, false]
  b_pch                 true                 [true, false]

Compiler options:
  Option                Current Value Possible Values
  -----
  cpp_args              []
  cpp_link_args         []
  cpp_std               c++11                [none, c++11, ...]

Directories:
  Option                Current Value
  -----
  bindir                bin
  includedir            include
  libdir                lib/x86_64-linux-gnu

Project options:
  Option                Current Value Possible Values
  -----
  frobnicate            false                [true, false]
  subproj:opt           a                  [a, b, c]
```

The output splits the options into logical groups and prints them one by one. The first set of options are the *core options*. They are the same in all Meson

projects and deal with general project setup.

The next set are the *base options*. They are also fundamental options but they are either language or system dependent. That is, some base options only appear when building with certain toolchains. The first option listed is `b_lto` and it can be used to enable *link-time optimisations*. If no toolchain currently in use supports link-time optimisations, the option does not exist. This is the case when compiling, say, a pure Java project.

All base options' names start with the prefix `b_`. This prefix is preserved to based options. Trying to define a project option whose name begins with `b_` is an error and will cause Meson to exit immediately. There are other such preserved prefixes. Each supported programming language preserves its own language name as prefix, so for example all options that begin with `c_` are reserved for Meson's internal use for options related to the C toolchain.

This project is implemented in C++ and the next set of options are the language options. They all start with `cpp_`. Of particular note is the `cpp_std` option that sets the version of the language in use. In practice many projects try to set this on their own with custom compiler flags, but that is not reliable as compiler flags set inside build files are not used, for example, in system introspection tests.

The next set is the *directory options*. They are identical in behaviour to core options but they are only used for defining installation directories. Thus they are listed in their own group for clarity.

At the very end are project options. They are as defined in the project's option file. Subproject options are also listed here, and their option names are prefixed with the subproject name followed by a colon.

Setting values of variables is almost anticlimactically simple. It is done by passing a `-D` command line argument to the configuration command.

```
$ meson configure -D buildtype=debugoptimized
```

This command produces no output. In fact it may seem like nothing has happened but the value has been updated. You can verify it yourself by running `meson configure` again and checking the value. The next time a build is started, Meson will detect the change in option values and will reconfigure itself with the new values before running the build.

10.5 Sharing options between projects

As was discussed in Chapter 6, subprojects are run in isolated sandboxes. They can not communicate with each other with the exception that the master project is able to access finished subprojects' variables. This presents a problem for

configuration because sometimes you want to ensure that the various options in subprojects are in sync. Suppose you have many projects all of which support many backends. When used on their own they can use any backend they wish, but for a conglomerate project you'd want all of them to use the same backend. There are also cases where it would be convenient, but not mandatory, for all projects to have same project options.

The latter of these is conceptually simpler as it reuses the `default_options` mechanism from `project`. Both functions for invoking subprojects, `subproject` and `dependency` also accept a `default_options` keyword argument.

```
subproject('subproj',  
          default_options: 'optname=overridevalue')
```

These default options can only be used to override the subproject's options, not global options as they have already been set. If an option is defined both in the subproject's `project` call and in the calling project's default options, the latter override the former. A command line switch overrides both of these.

With this approach there is no guarantee that option values are the same in the various subprojects. Even if they were, there is no guarantee that they will remain the same. Each option value can be changed with `meson configure` independent of other values. Enforcing same values for options can not be done. Meson provides a different elementary operation for this case called a *yielding option*.

An option can be marked as yielding with the `yield` keyword argument.

```
option(..., yield : true)
```

The semantic meaning of yielding takes a fair bit of text to explain but is in the end conceptually simple. When a yielding option is defined in a subproject (but not in the master project), *and* there is an option with the same name in the master project, then *and only then* will the subproject option become a reference to the master project's option. If even one of the conditions does not hold, then the `yield` keyword argument is ignored.

Thus if a project is built on its own, it gets its own versions of all yielding options but if it is built as a subproject, it will use the master project's versions transparently. If there are two subprojects that both have a yielding option and the master project has the same option, all three will refer to the same option and thus have the same value. On the other hand if the master project does *not* have the option then each subproject will have its own, independent option. They will *not* be the same and there is no way to make them alias directly. Yielding goes in one direction only, from subprojects to the master project.

Chapter 11

Custom build steps

The build target commands Meson provides are convenient but they can only go so far. There are many build tasks that can't be done with them and trying to provide a ready made tool for every eventuality is not possible. Because of this Meson, and indeed almost every build system ever made, allows users to define their own custom build steps.

11.1 Generating data files

Let's examine how custom build steps work by creating a simple *asset pipeline* for images. This is a common task, especially when developing computer games. Typically graphical assets are stored in revision control systems in a different format than what will be used in the final game. Typically files are kept in the drawing program's internal file format or in a general purpose file formats such as PNG. The deployed format on the other hand is usually heavily optimised for rendering. The actual format to use might depend on the deployment platform: mobile devices typically support different texture compression formats than desktop graphics cards.

The asset pipeline must first convert images from one format to another and then install the generated files. Suppose that we have a Python script called `image_converter.py` that can perform this task and that we have only one file, `image.png`, that needs to be processed and the file extension for the converted image is `.cmpr`.

The generation step is declared with the `custom_target` command. The `meson.build` file that implements the specified pipeline has the following contents.

120

```

project('imageconverter')

conv = find_program('image_converter.py')

custom_target('img',
    input: 'image.png',
    output: 'processed_image.cmpr',
    command: [conv, '@INPUT@', '@OUTPUT@'],
    install: true,
    install_dir: get_option('datadir') / 'myprog')

```

This declares a custom target that runs the conversion command. The `command` keyword argument specifies the actual command that needs to be run to convert the file. It does not take the input and output files directly as arguments but instead the weird looking strings `@INPUT@` and `@OUTPUT@`.

Meson will expand these variables internally to point to the actual files. The end result is that during compilation the following command will be executed:

```

python3 path/to/image_converter.py \
    path/to/image.png \
    path/to/processed_image.cmpr

```

This design choice was made so that build definition files do not need to know the actual output locations of the generated files. In fact the build definitions *can't* know that, since the layout of the build directory is chosen by the backend. This does place an additional burden on file generators: they can't choose where and how to lay out their own files, they must conform to the setup passed to them via command line arguments.

After the sample project's build directory is set up, the project can be built:

```

$ ninja
[1/1] Generating img with a custom command.

```

and then installed:

```

$ DESTDIR=/tmp/stage ninja install
[0/1] Installing files.
Installing processed_image.cmpr to /tmp/stage/usr/share/myprog

```


11.2 Dependency files

121

An important aspect of custom build steps is *reliability*. Whenever any input or script files change, the output files must be regenerated. Meson sets up these dependencies automatically. But if we slightly change the way the program works, things no longer work as expected. Instead of converting a single image let's assume that the task is to take many different images, combine them in a single big image and then compress the result. Joining images in this way is called a *texture atlas*.

Combining individual images can get fairly complex and writing all that logic in the build file as arguments to the compressor script is not fun. Instead we write the combination rules to a file and pass that as an argument to the combiner script. A simple JSON file describing how to generate a 2×2 composite image configuration file might look like this:

```
[
    ['file1.png', 'file2.png'],
    ['file3.png', 'file4.png']
]
```

and it would be invoked with the following Meson snippet.

```
custom_target(...,
    input: 'composite.json',
    output: 'processed_image.cmpr',
    command: [comp, '@INPUT@', '@OUTPUT@'])
```

This generates the output, but it is not reliable. If any of the source images is changed, Meson will not regenerate the composite image. This seems like a fairly big problem, the only two ways to solve it seem to be to either parse the contents of the JSON file and try to detect file names or manually list all files this target depends on. These approaches require either understanding the contents of arbitrary input files or manual work, respectively. Neither of these is a workable solution except in the simplest of cases.

This same problem comes up in regular compilation all the time: source files must be recompiled when they, or any header they include, changes. This problem was solved in the early 90s by having the compiler write out a *dependency file* which lists all header files that it processed while compiling the current file. The build invoker then reads these files after each successful compilation and stores the dependency information for the next compile.

The de facto compiler driver at the time was Make, so the format of the dependency files was chosen to be a Makefile subset. Any program that can

generate a dependency file in this format can be integrated with Meson to get automatic and reliable dependency tracking. The format itself is simple. If we add support to the generator script, its dependency file output would look like this:

```
processed_image.cmpr: ../composite.json \  
  ../image1.png \  
  ../image2.png \  
  ../image3.png \  
  ../image4.png
```

The file format is straightforward. The first entry is the output file followed by a colon followed by all files that were processed. The entire file could have been written on one line but using backslashes is a recommended best practice as it increases readability and thus debuggability. Having `../composite.json` in the file is not strictly necessary as Meson adds a dependency on all input files, but the script can not know what the build system above it is doing so it adds all processed files just to be sure. There is no downside to this duplication.

Once the generator script has been updated to write dependency files, we need to tell Meson about it.

```
custom_target(...  
  command: [comp, '@INPUT@', '@OUTPUT@', '@DEPFILE@'],  
  depfile: '@OUTPUT@.d')
```

Dependency file integration is enabled with the `depfile` keyword argument which specifies the name of the dependency file. By convention this is the output file name with an extra `.d` suffix. This ensures that the dependency file names are unique so different targets won't accidentally overwrite each others' dep files.

Most programs need to be told where to write the output file. In this simple script it is simply listed as the third argument. The text string `@DEPFILE@` will be expanded to the actual file name just like `@INPUT@` and `@OUTPUT@`. For actual scripts you may need to write extra command line arguments such as `['--dep-file', '@DEPFILE@']`.

11.3 Special strings in command arguments

Meson will automatically substitute the following template strings in command arrays.

@INPUT@ expands to a full path to the input file. If there are multiple input files, this entry gets expanded to all of them. So for example a command array

[exe, '@INPUT@'] and two input files would generate a command array [exe, 'sub/file1', 'sub/file2']. The expansion does *not* happen if the entry has any other text, such as '-i@INPUT@'. This definition is illegal when there are multiple output files but works if there is only one output.

@OUTPUT@ behaves identical to **@INPUT@** but works on output files.

@INPUT<NUMBER>@ expands to the output file at the index specified by the number. **@INPUT0@** refers to the first input file, **@INPUT1@** to the second one and so on.

@OUTPUT<NUMBER>@ behaves identical to **@INPUT<NUMBER>@** but expands to output files instead.

@OUTDIR@ expands to the directory where the output files should be written. A typical use case would be [cmd, '--outdir', '@OUTDIR@'].

@DEPFILE@ expands to a full path to where the script should write its dependency info file.

@PLAINNAME@ expands to the input file name without path segments, so a value like subdir/file.ext would expand to file.ext.

@BASENAME@ expands to the input file name without path segments, so a value like subdir/file.ext would expand to file.

11.4 Generating source code

Generating data files is fun and all, but things get *really* interesting when you start generating source code and compiling the result in other build targets in the same project. It is even possible to compile the generator program, generate source code with it and compile the result within a single project.

A typical source code generator problem is embedding assets inside an executable, also known as *resource bundling*. The idea is to have your binary files available in the source code as byte arrays. They are then immediately available without needing to look up resources on the file system. Here is how you'd use it.

```
extern const char *asset_array;
#include<stdio.h>

int main(int argc, char **argv) {
    printf("%s\n", asset_array);
}
```

124

```
    return 0;
}
```

The program will print out a text message that is in the `asset_array` variable. The actual text to be printed is in a file called `asset.txt`. This file needs to be converted to a C source file defining the array and compiled in the target. This is achieved with the following Meson build file.

```
project('c_gen', 'c')

srcgen = find_program('srcgen.py')

generated_c = custom_target('c generation',
    input: 'asset.txt',
    output: 'generated.c',
    command: [srcgen, '@INPUT@', '@OUTPUT@'])

executable('prog', 'prog.c', generated_c)
```

The only difference to previous file generator files is that the return value of `custom_target` is captured in a variable and then added in the executable target's list of sources. Meson will set things up automatically, including all dependencies so that the build runner will first run the generation step and only *after* it has finished, does it try to compile the result. The actual build is uneventful producing a runnable executable that prints the contents of the asset file.

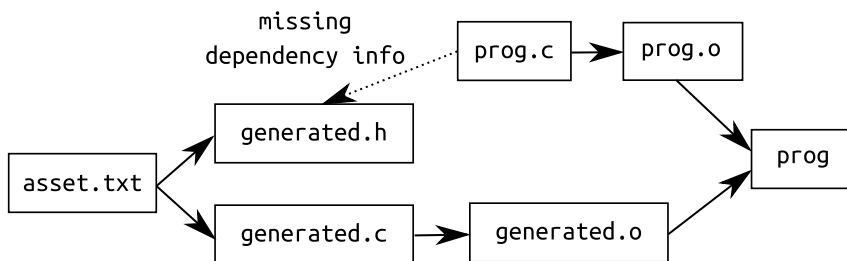
```
$ ./prog
Asset text!
```

11.5 Generating source and headers

The notable downside in the approach given is that you need to write the array definition in `prog.c` manually. The standard approach would be to `#include` a header with the definitions instead. We can change the generator program to also create the header, and thus the program source looks like this:

```
#include<generated.h>
#include<stdio.h>

int main(int argc, char **argv) {
```



125

Figure 11.1: A broken build graph, `prog.c` has not been declared to depend on the generated header `generated.h` leading to a build failure.

```

    printf("%s\n", asset_array);
    return 0;
}

```

Only the first line has changed. If we try to compile this program, there is an unexpected problem:

```

$ ninja
[1/4] Compiling C object 'prog@exe/prog.c.o'.
FAILED: prog@exe/prog.c.o
<<compilation command>>
../prog.c:1:9: fatal error: generated.h: No such file
#include<generated.h>
    ~~~~~
compilation terminated.

```

Even stranger is that if you run the build command again, it succeeds:

```

$ ninja
ninja: Entering directory `build'
[3/3] Linking target prog.

```

This failure is caused by a combination of two things: the build definition is slightly broken and the Ninja backend is aggressively parallel. It tries to run as many things simultaneously as possible to minimise build times. To understand why the build fails, we need to look at the whole build graph for this project. It can be seen in Figure 11.1.

When Ninja parses its input file it generates an internal graph similar to this one. It then tries to launch as many build tasks as it can up to a maximum

number, which by default is the number of CPU cores on the current machine plus two. Because the build definition is broken, Ninja sees two tasks that don't depend on anything: generating the source files from `asset.txt` and compiling `prog.c` to an object file. It launches both of these at the same time leading to compile failure since the header does not exist yet. Ninja will wait for the generator script to finish, though. Thus when Ninja is run again, the header exists and the build succeeds.

This failure mode is *nondeterministic*. If Ninja launched the generator script first and would get suspended by the operating system in just the right way, the generator could successfully run before the compilation process starts. These kinds of sporadic failures are strange and hard to debug unless you know what causes them. These errors are roughly analogous to *race conditions* in multi-threaded programming.

The proper solution to this problem is to add the missing dependency information in the build definition. Meson has one rule for this: if a build target has generated headers in its sources, then it will ensure that *all* headers are generated before *any* source belonging to the target is compiled. Thus it is the responsibility of the build definition writer to add generated headers in the source list of all targets that use them.

We need to change the build definition slightly to achieve this. The generator command needs to be changed to create two output files and then add that to the target.

```
project('c_gen', 'c')

srcgen = find_program('srcgen.py')

generated = custom_target('c generation',
    input: 'asset.txt',
    output: ['generated.c', 'generated.h'],
    command: [srcgen, '@INPUT@', '@OUTPUT@'])

executable('prog', 'prog.c', generated)
```

With this change the project builds without errors.

```
$ ninja
[4/4] Linking target prog.
```

This approach works well for single targets: all you need to do is to add everything in the target and things work. It does not work at all for the case when you need to use the headers from multiple targets but build the source only

once. This happens when you build a library with generated sources and need to expose some of the generated headers to users of the library. More specifically, this is an issue if you generate *public headers* as opposed to *private headers*

We need to declare a dependency that contains the generated headers, a proper include path and the library to link against. The declaration in all its simplicity looks like this.

```
# Same behaviour as above, generates a .c file and an .h file.
generated = custom_target(...)
lib = library('name', generated, ...)
incdir = include_directories('.')
lib_dep = declare_dependency(link_with: lib
    sources: generated[1] # Only the header, not source.
    include_directories: inc)
```

The important piece here is the `sources` keyword argument. All entries in it will be placed in the source list of all targets that use this dependency. Then Meson will ensure that the headers are built before any source code that tries to use it.

11.6 Using generators

Custom targets work fine for single files and targets but sometimes you need to run the same conversion steps for many input files. This is where generators come in. They provide a richer, but slightly more complex way of generating files.

Suppose we have our own DSL for describing objects. These descriptions are stored in files with a `.dsc` extension and need to be processed with a `dsc` compiler to produce a source and header pair. We have a program that needs to generate more than one of these files to work. We could solve this by writing a custom target for each file pair, but with generators we can define the rule only once and use it on many input files.

The build definition using a generator looks like the following.

```
project('generator', 'c')

idl_compiler = find_program('dsc_compiler.py')

g = generator(idl_compiler,
    output: ['@BASENAME@.c', '@BASENAME@.h'],
    arguments: ['@INPUT@', '@OUTPUT0@', '@OUTPUT1@'])
```

```
generated = g.process('object1.dsc', 'object2.dsc')

executable('prog', 'prog.c', generated)
```

Building files with generators takes multiple steps. The first one is calling the **generator** function that creates a new generator. Its only positional argument is an executable that will be used to do the conversions. A generator can be thought of as a rule for how to convert input files to output files. In essence it is a custom target, but the files to process have been separated from the processing rule.

A generator's rule accepts the same substitution variables as a custom target and they behave in the same way. This example sets up a build rule that converts a file called **object.dsc** into **object.c** and **object.h**. We then use the rule with the **process()** method and give it two input files. This will produce a total of four output files: two headers and two source files.

Note that up to this point *no files have been generated*. We have only set up a rule specifying how and which files should be generated. They only spring into existence when we use the return value of **process** in a build target. Only then will the files be generated, and they will be placed in the target's private directory. If the return value is used in several targets, each one of them will get their own generated copies.

The main program that uses these files merely generates and deletes one of each object type:

```
#include<object1.h>
#include<object2.h>
#include<stdio.h>

int main(int argc, char **argv) {
    object1 *c1 = new_object1();
    object2 *c2 = new_object2();

    printf("Objects created.\n");

    delete_object1(c1);
    delete_object2(c2);

    printf("Objects deleted.\n");
    return 0;
}
```


This is all that's needed, compiling and running the result is done in the usual way:

```
$ ninja
[6/6] Linking target prog.
$ ./prog
Objects created.
Objects deleted.
```

129

11.6.1 Extra arguments in generators

There are cases where you'd want to use the same generator rule for two different sets of inputs but they need slightly different command line arguments. This would mean that you need to define two different generators, one for each set of arguments. Meson provides a way to get away with only one definition. This is done with the argument string `@EXTRA_ARGS@`. When the `process` method is called, you can specify additional command line arguments with the `extra_args` keyword argument. If no arguments are specified the extra argument string is removed completely.

This means that if you have a generator like this:

```
g = generator(...,
    arguments: ['@INPUT@', '@OUTPUT@', '@EXTRA_ARGS@']
```

then calling it without extra arguments like this:

```
output = g.process('file1.in')
```

results in a command invocation like this:

```
generator_prog ../file1.in file1.out
```

Using extra arguments like this:

```
output = g.process('file2.in',
    extra_args: ['--something'])
```

yields this invocation:

```
generator_prog ../file2.in file2.out --something
```


Chapter 12

Cross compilation

Thus far we have focused only on *native compilation*. That is, building libraries and executables that are meant to be run on the current system, or one similar to it (same processor type, operating system and so on). In contrast this chapter is about *cross compilation*, where the result of compilation is meant to be used on a completely different machine.

There are many reasons for doing cross compilation rather than native. Perhaps the biggest one is resources. Embedded computers systems have very slow processors, only a few kilobytes of memory and no disk. Running a compiler on such a machine is simply not possible, and if it were, the processor is so slow that compiling even a trivial program would take hours.

The second major reason for cross compilation is *bootstrapping* new processor architectures. In this case code can not be compiled natively since there is neither a compiler to run nor a system to run it on. New platforms are bootstrapped by cross compiling the bare minimum set of programs needed to compile code natively on the device. Then the rest of the system can be built natively with this system. Bootstrapping new platforms happens fairly rarely, but being able to do that is important.

Cross compilation usually targets a different processor, but this is not a requirement. It is also possible to cross compile code on the same processor architecture but for a different operating system. A common case is compiling Windows applications on a Linux machine using the MinGW cross compiler toolchain. This allows one to develop software on two different platforms on one machine simultaneously rather than needing to change computers or use virtual machines.

The basic principles of cross compilation are fairly clear, but unfortunately there is one major hindrance.

12.1 A word about nomenclature

132

The terms used in cross compilation are *confusing*. Some words are used in unexpected ways. There are also terms which mean different and opposing things depending on which terminology is being used. Sometimes sentences are confusing and illogical because of this. Sometimes sentences are confusing and illogical on their own. This is made even more problematic by the fact that some books, articles and web sites have errors where they first define their terms and later on use them incorrectly.

Because of this, learning about cross compilation can be confusing at first. You may feel confused and lost in a maze of words that make no sense. You might be tempted to think cross compilation is something you can't possibly understand and just give up¹.

If this happens to you, just remember: *Don't panic!* Understanding cross compilation takes some time and pondering. If you keep working on it, even if some parts seem ununderstandable, eventually something will click inside your head and the concept will become clear.

With that out of the way, let's dive in. The biggest source of problems and confusion is that in cross compilation there are not two different machines, but actually three.

build machine The computer used for running the cross compilation jobs. This is usually the developer's desktop or laptop machine.

host machine The computer where the output of the cross compilation will be run on.

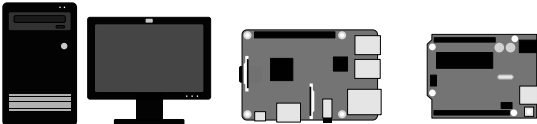
target machine The computer where the output of the compiled program will be run on.

This way of describing cross compilation is called the *Canadian Cross* and is the native setup of e.g. the GNU toolchain. In some texts these machines are called just "build", "host" and "target", but that increases the potential of mix-ups. In this book we will always call them with their full name for clarity. The first two types are as discussed above, but the target machine is where things start to get complicated.

12.2 A practical example

Rather than try to work this out via abstract thinking, let's look at a practical example instead. Suppose you are working on a project that runs on a small form

¹The author of this book admits to feeling exactly like this when he first encountered this issue.



Platform	Desktop PC	Raspberry Pi	Arduino
Processor	x86_64	ARMv7	AVR
Operating system	Linux	Linux	None

Figure 12.1: The three machines used in the cross compilation example.

factor Linux computer such as the Raspberry Pi. This is a popular approach to building devices that talk to external hardware. Its main limitation is that it does not have the hardware for real time operations with precise timing requirements. An example of such a task is controlling servo motors on drones.

One solution to this is to add an isolated and dedicated computer whose only job is to manage the real time task. In this example we're going to use an Arduino, which is a widely used line of hobbyist microcontroller boards. Just like any other computer it needs software to run. Microcontrollers don't have an operating system but instead they run a single program. These kinds of low-level programs are called *firmware* and the concept of running a program directly on the hardware without an operating system is called *bare metal*. Including the desktop PC we now have three different computers in this project. The three machines are detailed in Figure 12.1.

Typically firmware images are loaded onto boards and then left to run and never touched. Our case is different, and we want to be able to build and deploy new firmware on the microcontroller on the fly, and further we want to be able to generate the firmware images on the Raspberry Pi rather than providing prebuilt firmware images². Creating new firmware requires two things: the firmware's source code and a cross compiler that can generate firmware images to run on the microcontroller's AVR processor.

The Raspberry Pi has an ARM processor, so the firmware compiler must be a cross compiler (as its output can not be run on the Raspberry Pi itself). This compiler is a version of the GNU compiler called *avr-gcc*. Most Linux distributions provide a prepackaged version of this compiler but that version is too old so we want to build our own to get the very latest compiler version. This is where we get into trouble.

GCC is a *big* program. Compiling it on the Raspberry Pi can take hours. Since we have a fast multicore desktop machine available we'd want to compile

²Why? Because we can. And because it is an interesting challenge.

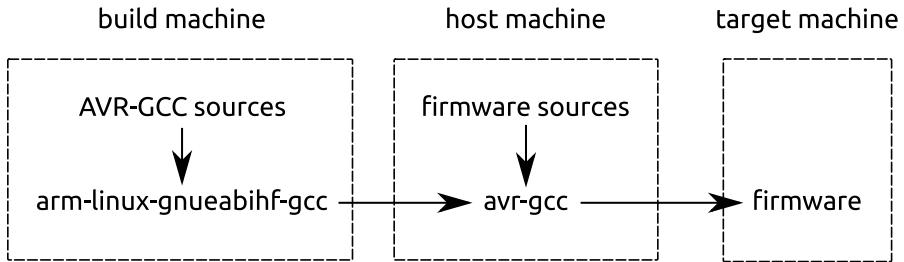


Figure 12.2: A full canadian cross setup for cross compiling and using a cross compiler.

the cross compiler with it. In order to do that we need a cross compiler that compiles from a desktop machine to the Raspberry Pi. This compiler follows the standard GNU naming scheme and is called `arm-linux-gnueabi-hf-gcc`. The leading part is called a *GNU triplet* and it uniquely defines the platform the compiler is meant for. This particular triplet has three pieces information: the processor (arm), the operating system (Linux) and the ABI (gnueabi-hf). This string can have up to four pieces of information, which may seem strange for a concept called “triplet”, but there you go. This act of cross compiling a cross compiler is the most complicated cross build scenario there is because all the three machines are different.

The *build machine* is the desktop PC with a standard x86_64 processor. The *host machine*, where the output of the compilation will be run on is the small Linux computer with a 32 bit ARM processor. Finally the *target machine* is where the generated firmware image will run. It is the bare metal AVR processor on the microcontroller board. A data flow diagram for this can be seen in Figure 12.2.

Note that there are two different cross compilation phases in this setup. The first one is the above mentioned cross compiling of the cross compiler. The second one is cross compiling the actual firmware. Their cross build setups are different:

	build machine	host machine	target machine
compiler	x86_64	ARM	AVR
firmware	ARM	AVR	AVR

12.3 Other naming setups

The Canadian cross is not the only way of categorising cross compilation setups. The most common other way ignores the target machine completely as it is so seldom used. Unfortunately they *also* change the meaning of *host* and *target*. In this setup the host machine is “the machine the compiler is running on” and the target machine is “the machine the result will run on”.

Thus the Canadian cross’s build machine is called the host machine with this nomenclature and correspondingly the host machine is the target machine. This is all terribly confusing but sadly the world of cross compilation has not converged on one generally accepted set of terms.

135

12.4 Cross compilation with Meson

A fundamental difference between native and cross compilation from a build system’s point of view is one of discoverability. When compiling natively every thing you’d want to find out about the platform can be easily looked up and tested. In addition most things are standardised and compilers have well defined names. With cross compilation none of this is necessarily true.

Since the information can not be automatically detected, it must be provided to the build system somehow. In Meson this is done through a mechanism called a *cross file*. It is a plain text file that looks very much like the standard INI file used by many programs and operating systems.

A sample cross file for compiling to a 32 bit ARM Linux device looks like this.

```
[binaries]
c = '/usr/bin/arm-linux-gnueabi-hf-gcc'
cpp = '/usr/bin/arm-linux-gnueabi-hf-g++'
ar = '/usr/arm-linux-gnueabi-hf/bin/ar'
strip = '/usr/arm-linux-gnueabi-hf/bin/strip'
pkgconfig = '/usr/bin/arm-linux-gnueabi-hf-pkg-config'

[properties]
c_args = ['--system-argument']

[host_machine]
system = 'linux'
cpu_family = 'arm'
cpu = 'armv7'
endian = 'little'
```

There are three main sections in a cross file. The first one is the *binaries section*, which lists the paths to all binaries that will be used for the cross compilation. In this example the file specifies both a C and a C++ compiler, the static linker `ar` as well as helper tools `strip` and `pkg-config`.

The second section is the *properties section*, whose purpose is to provide free form information for the cross build. In this file the `c_args` option means that the `-system-argument` should be used for every C compiler invocation. There are other things you can specify here that will be introduced later in this chapter. In addition to system data you can also specify arbitrary key-value pairs here and query them from your Meson build files.

The final section is the *cross machine definition section*. It consists of a host machine definition or a target machine specification or both (when cross compiling a cross compiler as discussed earlier in the chapter). This file specifies only the host machine, meaning the target machine is implicitly assumed to be the same as the host machine.

Cross compilation is set up by using the `--cross-file` command line argument when setting up a build directory for the first time. Whether any given build directory contains a native build or a cross build is set on the first invocation. It can not be changed after the fact. Assuming the cross file is called `arm32-cross.txt`, the Meson invocation to use it would look like this:

```
$ meson --cross-file=arm32-cross.txt builddir
```

Cross files are usually unique to each build machine, but not always. The only strictly platform dependent part of it is the list of executables. If two platforms use programs that have the same names and are located in the same paths, then the same cross file should work on both. Otherwise the file needs to be modified to match the setup of the build machine.

12.4.1 Machine properties in cross files

There are four different pieces of information that you must specify in the cross file machine definitions. The values for all of them are plain strings, they must all be present for the cross file to be valid. All values specified should be in lower case only. This book lists only the most common values of these options, a full up-to-date list of possible values can be found on the Meson web site.

The first one of them is *system*. It contains the name of the operating system running on the the machine, such as `linux`. This is mostly self-explanatory though there are a few surprises in the naming due to compatibility with other build systems. All versions of Windows have the value `windows` and both macOS and iOS have the value `darwin` after the kernel used on those platforms. Systems

that don't have an operating system, such as firmware or embedded, should set this to `baremetal`.

There are two different properties for identifying the processor: `cpu_family` and `cpu`. The reason for this is that sometimes you don't care about the specific details of the processor, just its general type. In contrast other projects require as much information about the processor as possible. These include compilers and assemblers and also projects that want to use CPU instructions that depend on the microarchitecture. If Meson did not provide both of these properties, many projects would need to have ad hoc code to determine one or the other.

In practice most processors have the same value for `cpu` and `cpu_family`, for example all 32 bit Mips processors identify as `mips`. Notable exceptions are 32 bit ARM processors and x86. The latter's `cpu` values vary wildly from `i586` to `x86` and others.

The final piece of data is the endianness of the platform and it can have either the value `big` for big endian or `little` for little endian. There are processors with other endiannesses, but thus far nobody has needed to compile code for them using Meson so they are not supported. Given that processors using any other endianness than big or little were last created in the 70s, it seems unlikely that support for those will ever be added.

12.5 Cross file lookup

When people get started with cross compilation they typically create a cross file for each project and add them to revision control. This works for projects with few developers but it becomes problematic when the number of contributors and platforms increase. Different platforms may have different versions of various toolchain binaries, they might be in different locations and so on. In these cases it would make sense to provide cross files for each platform rather than for each project. In this way one single cross file could be used for any project.

Meson supports this workflow by looking up cross files from the file system using a standard method called the *XDG Base Directory Specification*. The actual specification is fairly involved, but for our use case we only need to know that there are two environment variables, `XDG_DATA_DIRS` and `XDG_DATA_HOME`, that specify where applications should look up system and user data files. The former location is set up by the system administrator and contains global directories like `/usr/local/share`. Meson will try to look up cross files in the `meson/cross` subdirectory of the directories specified in the environment variable.

The other environment variable is used to look up cross files in the user's home directory. Typically this variable either points to some system-dependent dir under the user's home directory or is unset in which case Meson will use the default value of `~/.local/share`. Just like above the `meson/cross` subdirectory

is used to house the cross files. In practice what this means is that if you create a cross file with this name:

138

```
~/.local/share/meson/cross/mycross.txt
```

Then you can use this file to cross compile any project by setting up your build dir with the following command:

```
$ meson builddir --cross-file=mycross.txt
```

12.6 Multiple cross files

Cross files can be layered on top of each other by specifying them on the command line:

```
$ meson --cross-file=cross1 --cross-file=cross2 ...
```

Entries in later cross files override earlier files. Thus if **cross1** specifies both a C and a C++ compiler and **cross2** specifies only a C compiler, the build would use **cross2**'s C compiler and **cross1**'s C++ compiler.

Typically one defines a “base” cross file that has all the common toolchains and platform properties and then would add individual override files for other features. An extensive CI system might need to run cross compilation test suites for ten or even more different versions of any given compiler. This layering approach ensures that all the common platform features remain the same on all these build setups.

12.7 Constants in cross files

As cross files get bigger they usually contain repetitive information, typically in compiler arguments. For example you might need to pass some specific piece of information about the underlying platform or chipset to many different tools. This may cause typos or copypaste errors leading to bugs that are difficult to track down. To simplify these cases you can specify constants in a special section in your cross file and then use the variables in the rest of the cross file.

```
# Note that the constants section must always be the first thing
# specified in the cross file.
[constants]
arch = 'aarch64-linux-gnu'
clike_args = ['--sysroot=/opt/toolchain' / arch,
```

```
'-DSYSTEM=' + arch]
```

```
[properties]  
c_args = clike_args  
cpp_args = clike_args
```

139

In this example we first specify the architecture and then the compiler arguments needed for the cross compilation. The first argument is built with the `/` operator so Meson will join the two as file system paths whereas the second uses the `+` operator resulting in basic string concatenation. If you have multiple toolchains of a similar type, creating cross files for them is as easy as copying the file to a new name and changing only value of the `arch` variable.

12.8 Native files

Since cross files are so convenient at describing all cross compilation properties in one definitive place, it would seem reasonable to have similar functionality for defining native builds as well. Meson provides this and the corresponding files are called *native files*. They are used just like cross files:

```
$ meson --native-file=basic.txt ...
```

The syntax for native and cross files is identical, with the exception that you can only define the build machine in the native file and host and target machines in a cross file. A build can use both file types:

```
$ meson --native-file=basic.txt --cross-file=cross.txt ...
```

This is convenient for build reproducibility. Putting all platform configuration in these files means that you can easily rebuild the software without fear that some configuration data, typically environment variables, has been lost.

12.9 Running tests when cross compiling

Even though cross compilation is a fairly complicated affair, it turns out that for build systems cross compilation is identical to native compilation. It can be seen as just like using a different compiler with potentially different compiler flags. The differences between the two happen both before and after the actual compilations. The former is about configuration and the latter is about running the compiled programs.

Native programs can be run directly, but cross compiled programs generally can not. Programs compiled for one processor architecture can't run on a different one. For example, ARM binaries do not work on an x86_64 processor. There are cases where this is possible, for example on a *multi-arch* Linux distribution you can run 32 bit x86 applications on a 64 bit x86_64 processor but that is the exception rather than the rule. This is a problem because it means you can't run the program's tests to verify that it is working.

Even though cross compiled programs can't be run directly, sometimes they can be run via helper applications.

System emulators such as QEMU[2] emulate a complete computer system including the processor and other hardware. The program is then executed on this virtual system. This approach is easy to use but its major downside is that hardware emulation can be $10\times - 1000\times$ slower than running the program on real hardware.

System reimplementations are not emulators, but instead a reimplementation of "system software" of some other operating system. Perhaps the best known example is the Wine[1] compatibility layer that allows you to run Windows applications on platforms such as Linux and macOS. This approach requires that both systems use the same processor type.

Other methods are also possible. One could transfer the binaries to a different machine, run them there and then transfer the results back. This could also be done by mounting a shared network drive on both machines, and running compile steps on the local machine and tests on the remote machine.

Meson's abstraction for all of these is the *exe wrapper* introduced in Section 8.3.4. A wrapper can be defined in the cross file like this:

```
[binaries]
...
exe_wrapper = 'wine'
...
```

This declaration tells Meson that cross compiled applications can be run by prefixing them with the given exe wrapper. Thus if Meson would need to run a cross compiled program like this:

```
./some_exe --argument
```

It would transparently run it like this instead:

```
wine ./some_exe --argument
```

For Wine this is all that is needed to run the full test suite. Some other tools may require command line arguments. Those can be specified by defining the exe wrapper as an array.

141

```
[binaries]
...
exe_wrapper = ['emulator', '-arg1', '-arg2']
...
```

If the setup is more complicated than this, the smart thing to do is to write a script file that sets up everything needed to run the programs. A shell script for this might look like the following:

```
#!/bin/sh
# Set up the environment here.
emulator-command "$@"
```

Where "\$@" is a Unix shell expression meaning “take all arguments given to this script, quote them and use the result as arguments to the emulator command”.

12.10 Cross compilation and code generators

Most projects don't require extra work to be cross-buildable unless they both generate source code during compilation *and* build the executables and run them as part of the build. Since cross compiled binaries can usually not be run directly, this is a problem. A traditional solution to this problem is to make the build a two phase process. First all the tools are built with the native compiler and put somewhere where they can be found. Then the code can be cross built using these tools.

This approach works but it is a bit cumbersome. If you edit the code generators' code you first need to build and install them and only after that run the cross compilation. Some build systems might not pick up changes in system tools and thus not regenerate all built sources. It would be more convenient if the build system could take care of all of this in a single build invocation.

Meson provides two different solutions for this problem. The first one is based on an exe wrapper as introduced in the previous section. When an exe wrapper is set, Meson will use it transparently when built executables are used during the build. The end user does not need to do any changes to the build files.

This leaves the third and the trickiest case: when cross compiled executables can not be executed. This case is solved by tagging an executable as *native*, like this:

```
executable('srcgenerator', 'srcgenerator.c',
           native: true)
```

When cross compiling Meson always sets up two different toolchains, one for cross compilation and one for native compilation. The latter is the toolchain that would be used if the project were being compiled natively. Tagging the target as **native** tells Meson to build it with the native compiler, which means that it can be run as part of the build. The **native** keyword can be found in some other functions such as **dependency**.

Building and using code generator tools typically follows this pattern:

```
dep = dependency('depname')
if meson.is_cross()
    native_dep = dependency('depname', native: true)
    gen = executable(...,
                    native: true,
                    dependencies: native_dep)
else
    gen = executable(...,
                    dependencies: dep)
endif
# Use gen to generate code here.
```

This call to **dependency** will return a native dependency when building regularly and a cross dependency when cross compiling. In the latter case we need to build the executable with the native compiler which also means we need to find the native dependency version. It is stored in the **native_dep** variable and passed to the **executable** call. This builds the program with the native toolchain. The **else** branch is only executed during native builds and builds the executable in the usual way.

In either case after the **if/else** block has been processed, the **gen** variable contains an executable that can be run as part of the current build, even without exe wrappers.

12.11 Firmware upload targets

Once a cross built executable has been built and tested it needs to be deployed to a machine. This step is highly hardware dependent and has been implemented

in a variety of different ways. The output files can be copied to a shared network drive or transferred directly to the device with e.g. `scp`. It is also possible to copy the files to a USB disk and use it to boot the device. All these approaches work fine, but they require an existing operating system (or at least a boot loader) on the target device.

Bare metal embedded devices do not have that. The firmware executable is the only piece of software that will run on the device. It must be installed on the device in a special way so that the system will start executing it immediately after power has been switched on. The most common solution is to have a piece of non-volatile memory called *flash ROM*, on the device whose only purpose is to hold the firmware executable.

Flash ROM is different from normal memory. It can not be written to directly from the device. Instead the system must be put in a special state so that the flash ROM becomes writable and then the firmware can be transmitted using a hardware specific protocol. This process is called *flashing* or *burning* the firmware and the program that does this is called, unsurprisingly, a *flasher* application. Nowadays flashing is usually done over USB with a command line flasher application provided by the hardware vendor.

The flasher application can be invoked by typing the command line by hand every time it is needed but it is much more convenient to create a Meson run target to do this instead.

```
firmware_bin = executable(...)
flasher_app = find_program(...)
run_target('flash',
    command: [flasher_app,
              # Other application arguments here.
              '--file',
              firmware_bin])
```

Flashing the device can be done merely by invoking the flashing target.

```
$ ninja flash
```

Since `firmware_bin` is in the command line arguments, Meson will automatically both expand the file name and add a dependency between the firmware target and the flash target. Invoking the flash target will always rebuild the firmware target if it is out of date, ensuring that a stale firmware binary is not flashed on the device by accident.

Chapter 13

The Wrap dependency download mechanism

Before dependencies can be used, they need to be obtained somehow. External dependencies like this can be installed with the system package manager such as *apt*. Internal dependencies are harder, because they require obtaining the source code and putting it in the `subprojects` directory in a very specific way. There are many ways to accomplish this.

The first one is called *vendoring*, where the full source code of the dependency is duplicated in the source tree and put into revision control as if it was self written code. A development style where all self written code and every dependency is stored in a single repository is often called a *monorepo*. This approach is fairly simple to implement, but there are also downsides. Duplicating code severs the connection to the original code. Unless developers are careful, it is all too easy to forget to keep vendored dependencies up to date. This approach is mostly used inside corporations.

In the other end of the spectrum are projects where vendoring is not only frowned upon, but prohibited altogether. This is the case for many free and open source projects. A requirement for getting a project in a Linux distribution such as Debian, Fedora or Ubuntu is that it *must not* contain any vendored code. All dependencies must come from the system. These kinds of projects must obtain their self built dependencies in some other way instead, especially when they are built on platforms that do not have a concept of system package management.

A common solution is to clone the dependency projects' repositories inside your project's source tree. This is called *submodules* in Git and *subrepositories* in Mercurial. Most other revision controls have similar functionality. The major downside of this approach is that cloning a repo brings in the entire development history, which can be slow when there are many large dependencies.

Meson has a builtin dependency downloader system to handle these and other ways of obtaining dependencies. It is called *Wrap* because it works by adding a small layer on top of upstream projects to make integration easier. This not entirely unlike wrapping a burrito in paper so you can eat it without making a mess.

13.1 The basic design

A Wrap dependency is nothing more than a file that specifies how a the source code for a dependency should be obtained when requested. The dependency resolution algorithm behaves in the following manner.

1. A project requests a subproject called `mydep`, either via the `subproject` function or indirectly via a dependency fallback.
2. Meson looks up if there is a file called `mydep.wrap` in the master project's `subprojects` directory.
3. If yes, it will be parsed and the dependency will be downloaded according to the instructions given in the wrap file.
4. If not, Meson will fall back to regular subproject lookup, that is, looking for an existing project in `subprojects/mydep`.

This is the only way to obtain source code via Wrap. You can only get an entire project and only in the `subprojects` directory.

This design makes it possible to use Meson both in monorepos and for isolated per-project repos. Meson itself does not care where the dependencies come from, just that they are stored under the `subprojects` directory in their own directories.

Wrap files (or “wraps”) themselves are simple plain text files in the INI format.

13.2 Downloading revision control checkouts

Wrap files are best understood through examples, so here is a Wrap file for a subproject that is used for testing Meson.

```
[wrap-git]
directory=samplesubproject
url=https://github.com/jpakkane/samplesubproject.git
revision=head
```

The section text `wrap-git` quite literally says that this is a Wrap file for checking out dependency source code with Git. The actual content consists of three key–value pairs.

directory is the directory name where the checkout will be written to. This defines a subdirectory directly under **subprojects**. The directory must have only one segment, that is, it may not contain `\` or `/` characters.

url is the download location. This is passed directly to `git clone` and can be in any format supported by Git.

revision can either be a Git hash which unambiguously selects the revision to check out, or it can be a Git tag, branch or the string `head`, which is an alias to the remote repository’s **master** branch.

Whenever Meson reconfigures itself (that is, when the build definition has changed), it will refresh all subprojects. Wraps that follow a specific branch will be updated to the latest upstream commit. This may be considered a security vulnerability. If the upstream repository has been compromised with malicious code, it will be automatically downloaded and used in the current project. Using a commit hash as the **revision** value makes this problem go away, but requires manual work to keep up to date.

13.3 Downloading a release archive

There are cases where you want to use a release for the dependency rather than the full repository. Typical reasons include legal and policy requirements. Following releases is also more stable than having your dependencies continuously change.

Releases also amplify a different kind of problem, which is that many dependencies use build systems other than Meson. Since you can only build native Meson projects as subprojects, this presents a problem of how to get your build definitions in the release. With revision control you can make your own branch with the files you need and tell Meson to track that repo.

The Wrap files for releases solve this issue by having two independent parts, the *upstream release archive*, often called a *tarball*, and an optional *patch file* on top of it. Here is a sample Wrap file for the Zlib compression library.

```
[wrap-file]
directory = zlib-1.2.8

source_url = http://zlib.net/fossils/zlib-1.2.8.tar.gz
```

148

```

source_filename = zlib-1.2.8.tar.gz
source_hash = <<SHA-256 hash code>>

patch_url = <<wrap file download path>>
patch_filename = zlib-1.2.8-8-wrap.zip
patch_hash = <<SHA-256 hash code>>

```

The `directory` entry has the same meaning as earlier but for release archives there is an extra requirement. The established practice is that releases should have all their files in a subdirectory having the name and version number of the current release. This ensures you can unpack them and not have to worry about accidentally overwriting an earlier version. The Wrap file's directory must match the directory name inside the archive.

The release archive is downloaded directly from upstream. The `source_hash` entry is the SHA-256 checksum for the file. Whenever Meson downloads the file it will calculate its checksum and verifies that it is the same as is written in the Wrap file. If there is a mismatch, Meson will exit with a hard error.

The entries starting with `patch_` are the same as above but they are for the patch archive. A patch archive contains only build definition files. Its contents will be extracted over the source tree creating a fully contained directory tree that can be built as a subproject.

13.4 Using the WrapDB

The Meson development community maintains a web service for downloading dependencies automatically. It is called the *Wrap database* or *WrapDB* for short and can be accessed at <https://wrapdb.mesonbuild.com/>. The service provides Meson build definitions even for projects that don't use Meson. These build definitions are contributed by volunteers.

The web page can be used for quickly browsing the available packages, but most interaction is done with the `meson wrap` command line tool. Suppose we have a test program that needs to use the Zlib compression library. First we search for it:

```

$ wrap search zlib
zlib

```

This command prints all available dependencies that match the search criterion, in this case there was only one. More info is available with the `info` command.

```
$ meson wrap info zlib
Available versions of zlib:
  1.2.11 2
  1.2.11 1
  1.2.8 8
  1.2.8 7
  1.2.8 6
  1.2.8 4
  1.2.8 2
```

There are multiple versions. The default command installs the newest version, which is what we want.

```
$ meson wrap install zlib
Installed zlib branch 1.2.11 revision 2
$ ls subprojects
zlib.wrap
```

The install command does not load any actual code, only the wrap file needed to download them when needed. The Wrap system also supports updates, which are usually initiated by checking for newer versions.

```
$ meson wrap status
Subproject status
  zlib not up to date. Have 1.2.11 2, but 1.2.11 3 is available.
```

In this case there is a new version available. The upstream version has not changed but the packaging revision number has increased from 2 to 3, indicating packaging fixes. The `update` subcommand can be used to bring the current version up to date with the latest upstream.

```
$ meson wrap update zlib
Updated zlib to branch 1.2.11 revision 3
```

On next build Meson will reconfigure the project to use the new dependency version.

Chapter 14

Converting an existing project to Meson

Most projects do not have the luxury of starting from scratch as they have existing code in production, which must keep working. This limitation applies also to build systems. Converting existing projects from an established system to something more modern can be a big undertaking, but it can also provide the biggest productivity payoffs. In this chapter we shall examine various aspects of converting a project from an existing build system to Meson. There are many different pitfalls and obstacles that pop up in many conversions, so we'll try to find workable approaches for them to make the conversion process as smooth as possible.

14.1 Why change build systems? Is it even worth it?

Before embarking on any rewrite or upgrade project, you should first ask yourself the most important question: *why*. Simply wanting to get the new shiny is not by itself a sufficient reason to start a major migration project. It may be harder to quantify the advantages for a build system compared to things like programming languages and third party libraries. In the end it's really not that different when you realise that build definitions are just code. The basic reasons and payoffs are mostly the same for changing build systems as they are for changing the “core” code.

14.1.1 Performance issues

The time it takes to compile a software project depends mostly on the compiler, thus the performance of a build system does not really matter as long as it can

spawn n parallel tasks on a machine with n cores. At least, that is a common belief many experienced people hold. This is both reasonable and expected. Unfortunately it is also completely false. Experience with porting projects from existing standard solutions shows that the speedups can be tens of percent for full builds and, amazingly, an order of magnitude for incremental builds.

A simple way to test the potential performance gains to be had is to do some measurements. The first one is the *no-change build time* which measures how long the build system takes to run when there is nothing to do. This should be instantaneous, even for projects with tens of thousands of source files. Anything more than about half a second should be a cause for concern. This might seem like an insignificant amount of time but it actually matters. If you need to wait for, say, five seconds every time you build, it causes disruption and brings developers out of “the flow”. This time is also a fairly good estimator of build performance in general. If no-change rebuilds are slow, most likely other parts of the system are slow as well.

The next test consists of making a trivial change in one source file and recompiling. This should not take much more than the amount of time you’d need to run the compile and link jobs by hand. If it does, then it implies a performance problem somewhere in the build system.

The final test involves making a change that forces the build system to fully reconfigure itself. This is typically done by editing one of the build definition files. It is harder to give target times for this operation, but for small projects it should be just a few seconds while huge projects might reasonably take tens of seconds or even up to a minute, especially if they do things like try to update dependencies over the network.

The final eyeball statistic you can use is to display the CPU load while doing a full build. Once compilation starts all cores should be working all the time until the end of the build. This may seem like something that should always happen but again there are many cases where it does not. The classical approach to writing Makefiles, so called *recursive make*, builds source code one directory at a time. Typically a single directory specifies one target. When recursive Make builds such a dir it can use all the cores when compiling individual source files, but linking is done with just one CPU. Only after linking has finished does it proceed to spawn processes in other directories.

14.1.2 Making changes is difficult

There are many custom build systems in the world. Some of them are written fully from scratch, while others consist of customisation code written atop an existing build system. Some of the largest code bases in the world are built with these kinds of tools, mostly inside corporate environments. When this works,

it can be an extremely powerful development tool. What seems to happen in practice is often quite different.

Usually these tools are written to replace an old system that is no longer working. The people doing the work are under stress to make the new system work as quickly as possible and do not have the luxury of planning and designing for the long term. This tends to lead to systems that solve the current problems quite well, but do not provide for future extensibility. By itself this would not be a problem. However as time passes the original developers transfer to other teams and even leave the company. Little by little the institutional knowledge of how to work the system gets lost and the people who can reasonably make changes to it get fewer and fewer. Extensive documentation would help to combat this issue, but most internal corporate projects tend to provide only scant documentation, if any.

Projects do not stand still, though. They become ever larger and need more features. In the worst case this causes the system to decay and end up with a situation where making any changes is extremely hard, so they are not done and as a result all developers on the project suffer. If, on the other hand, the custom solution is kept up to date, it incurs an ongoing maintenance burden. Whether this is beneficial or not depends on the use case.

14.1.3 Lack of documentation and integration

Perhaps the biggest problem of bespoke systems of any kind is that it can be difficult to find documentation on using them. In addition to the lack of internal documentation mentioned in the previous subsection, there is also the problem of external documentation. The more custom the system is, the harder it is to find help using external sources of information. Web search engines and discussion forums can be completely useless for debugging problems with uncommon or, in the extreme case, company confidential tools.

Similarly it is a lot harder to make custom tools work together with other tools. Most IDEs integrate well with the most common build tools, but making them work with custom ones might take some work and the end result is rarely as smooth as the one for popular tools.

14.2 Making sense of an existing build system

The first step in action is understanding. For legacy systems this might be more difficult than expected. Build systems that have gone unmaintained for a while have the tendency of becoming quite convoluted. A rough overview of such setups can be seen in Figure 14.1.

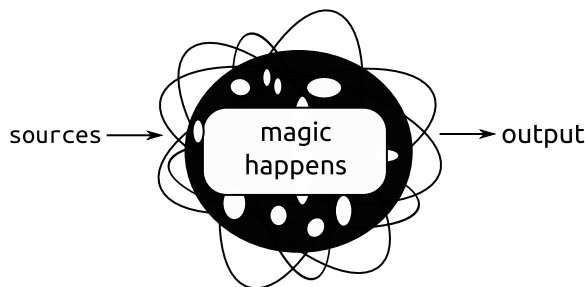


Figure 14.1: The architecture of a typical unmaintained build system.

In the worst case the existing build system may consist of tens of thousands of lines of custom code, possibly in an esoteric programming language. Deciphering what it is doing might take weeks of full time work. It is usually difficult to justify this amount of time to reverse engineering. It might be even harder to find a person willing to do it. Fortunately there is an alternative approach that can be used in almost all cases.

The unifying thing about build systems is that eventually they reduce to calling compilers and linkers to do the actual work. The command line interface for those tools is standard and fairly readable. All build systems also provide a way to get that information in text form. For example Make can be run in verbose mode so it prints all commands it runs to the terminal. Different Make implementations might have different ways of enabling it, but this information is readily available via documentation and Internet search engines. Once you know all the compiler flags the project uses to build its sources, reimplementing that in Meson is a lot simpler than trying to understand the innards of the existing system.

The same approach can be used in more general ways. The Meson source code ships with a file called `tools/ac_converter.py`, which can be used to convert GNU Autotools configuration checks to Meson. Autotools is implemented in a mixture of shell, Perl, Make and a special text processing language called `m4`. Configuration tests written in this system can not be reasonably machine parsed, they have to be executed. Fortunately the configuration check's main output is a file called `config.h`, which has a very rigid format. The converter script reads that file, detects what features the original script was trying to detect and generates the equivalent checks in Meson code. This typically saves a fair bit of work in conversion projects.

Many conversions benefit from custom conversion tools. Almost every build system has some sort of a human readable syntax to specify which sources go

into which targets. A simple text-based script can typically be used to extract most of this information and write it out in Meson form. This is doable and beneficial because the script does not need to be able to parse the entire file or even to be fully correct. Being able to translate 90% or more of the build definitions and fixing the rest by hand might make writing a custom converter script worthwhile.

14.3 Build tasks ordered by difficulty

As we have seen in this book, there are many different steps to take when going from source to a fully built binary. Some of them are more difficult to convert than others and each of them have their own quirks and tricks to be aware of. Below, we'll go through them starting with the easy ones and ending with the most difficult ones.

14.3.1 Target definitions

The bulk of most build files are target definitions. This includes things like which sources go where, what include directories and compiler flags they need and so on. Perhaps surprisingly this is usually the easiest part of any conversion. As discussed in the previous chapter, it may well be possible to create a script that does the bulk of the work. Even though this part is easy, it may be quite laborious and, at worst, tedious. Trying to do a one-shot conversion with just one person runs a high risk of developer burnout.

14.3.2 Configuration

Converting configuration steps is more difficult than target definitions, because they often contain custom code. Because of this it can not be automated to the same degree. Some human brain power is always required for these tasks. Fortunately most projects have noticeably fewer configuration steps than build targets.

14.3.3 Source generation

This is where things start to get tricky. Compiling executables that are then used to generate more source code becomes more and more common as project size grows. These cases can still be handled by following the guidance discussed in Section 11.4. A more problematic case happens with ad hoc source generation and manipulation steps. This is especially prevalent on projects that build with Make.

As the build steps in Make are basically shell invocations, it is tempting to embed source generator and manipulation commands inside Makefiles by using tools like Sed, Grep, Awk and Perl. These pipelines typically exist either to generate code or postprocess it to work around deficiencies and missing features of other programs.

It is usually recommended to get rid of these pipelines. The recommended approach is they should be rewritten in Python, which is usually faster and more performant, especially on Windows. If this is not possible or feasible, they should at least be extracted to standalone scripts. This cleanup operation can be done entirely in the existing build system before any other conversion work is started. It improves readability and the standalone scripts can be developed and debugged in isolation. Shell pipelines embedded inside Makefiles are only executed when the entire build is run, which makes for a slow and unpleasant development experience.

14.3.4 Data files

Generating and installing data files is similar to source generation with one important caveat: incorrect source generation is detected immediately as compilation errors but broken data or missing files can only be detected at runtime. If you are very lucky they are detected before getting shipped to end users.

Large programs may need an extensive set of configuration and data files to function. Worse, they might need to be in specific locations with specific names. As the old system has always put them in the right place, the code might not handle missing or malformed data files gracefully (as there has never been the need to check for this). This may very well lead to a case where the build succeeds and the program runs but then fails spuriously in a way that no-one can explain. It might even be that the issue is not in the data files, but is instead caused by some differences in compilation. In any case if this happens you can't really know why.

There is a fairly simple trick for debugging these problems. First you configure, build and install the program using the old system as usual. Then you delete all executables, libraries, plugins and other built code from the final install location. With this preparatory step done you take a fresh checkout of the code and remove all data file install commands from the new build definitions (if any had been written). Then you build it and install it *over* the previous install. This gives you an install that has all the binaries from your new setup but all data files from the old one. If issues still exist, you know that they must be in the compilation phase. If not, then the problem lies in the new build system's data file generation and install steps.

14.3.5 Build tree layout requirements

This case is particularly nasty, because it combines the previous three into one: an executable that is used to generate sources that requires many configuration files to build and which *must* be in specific directory paths to work. In the simple case the data files are already in the correct format and layout inside the source tree, but sometimes they are not. Meson is quite rigid in how files are laid out in the build dir. This is highly beneficial when switching between different Meson projects, because they all work in roughly the same way. The downside is that sometimes cases such as these can not be directly supported and require code changes in the project being converted.

Typically these sorts of requirements stem from the fact that existing tooling was not at the same level as it is now. For example the project might have its custom tooling to generate data from XML files. Rather than trying to convert these programs to conform to new requirements, it might make sense to rewrite them in a scripting language like Python instead. String manipulation and XML processing are usually a lot more pleasant to do in Python than most compiled languages. This has the added benefit that cross compilation becomes easier as the source generators do not need to be built during compilation, the scripts can instead be run directly.

157

14.4 Conversions involving an entire team

Small and medium projects can usually be converted in a single shot. As the project and team size grows, this is no longer possible. In addition to the sheer amount of work needed, other factors start to come into play. Developers must be trained to use the new system, mechanisms need to be put in place so that both the old and the new setup are kept working simultaneously and so on. This requires a coordinated and carefully designed transition plan.

Perhaps the most important piece is a working CI setup. If the project does not have CI, then setting one up takes priority over any build system conversion steps. Once a CI system is running it should be set to build both the code with both the old and the new build system. All additions to the main branch should be gated via this mechanism. This ensures that the conversion does not break when someone changes the build definitions of one system and forgets to do it on the other one.

The conversion should start at the lowest levels and proceed iteratively upward from there. In this way the parts of the project that have been converted can be built even though the end result does not fully work. Most projects have some kind of a foundational library that is used by most other pieces. The conversion should start with this and should be done either by one person or a small

team of experts. Typically these libraries may have unusual build requirements, source generation and are thus the most difficult ones to convert.

Once this is done the people who performed the conversion should gather the experience gained and write documentation. It does not need to be thorough but should cover at least the following:

- How the build is configured, compiled and installed.
- An outline of how a build target should be set up.
- How an existing build target should be mapped from the old definition to the new one.
- Description of all tools that have been created to help the transition.
- A list of typical problem cases that may come up, how to fix them and where to get help if problems occur.
- Any other useful tips and tricks as they are discovered.

With this the rest of the work can be split among the entire team. These tasks can be done inbetween other tasks during regular work or assigned via the usual tasking mechanism, be it scrum planning or Jira tickets. This approach has several advantages. It spreads the work to many people so the load on people running the conversion is reduced. Converting existing modules also serves as training on how to use the new build system and hands-on work is usually a better way of learning than just reading documentation. There is also less pressure on learning the new system as any mistakes won't cause problems in the end product. Final versions are still being built with the old system, which remains the same.

This phase of the conversion should be as short as possible. Needing to update both the old and the new system is taxing. Once all parts are converted, it is time for the moment of truth: validating that the output of the new system is functionally identical to the old one. This typically uncovers a bunch of defects in the new build setup. Once all of these are fixed, the build system conversion is finished and the new one can be taken into use.

The final step is deleting the old build system entirely from the system. The most popular approach is to make a major release and start the new development cycle by deleting the old build system. Others may choose to keep both the old and new ones around for one full release cycle as a safety precaution. Both of these choices are valid and choosing between them comes down to project requirements and desired level of risk aversion.

Chapter 15

A library sample project

Building and installing a library is simple but not sufficient for real world usage. In this chapter we look at all the additional work it takes to make a library that works seamlessly on multiple platforms and build setups. This may seem like a lot of work, but if you intend to provide a library that other people can use (which is what most libraries strive for), it is worth spending the effort. The easier it is for people to use your work, the more likely they are to adopt it for their own use.

15.1 Design requirements

The library itself will be called *Ducky* and it represents a very simple, object oriented concept of a rubber duck. There are three things you can do with a rubber duck object:

1. Create it.
2. Squeeze it.
3. Destroy it.

The actual requirements for our implementation are the following:

- The API should be in plain C.
- Provide a stable ABI.
- The implementation should be in C++.
- All internal implementation details should be hidden.

- Conform to all OS installation guidelines.
- The library must be usable both when installed on the system and when used as a subproject.
- The project should be usable both as a shared and as a static library.

15.2 The external API

A software project is usually created in one of two ways: either *inside-out* or *outside-in*. The former approach is the common one, where developers start working on a problem, writing code, and trying to understand the problem domain. After a while this produces a code base that solves the given problem. Then an API is created on top of this code and given out to other projects to build on. This approach works fairly well and is aligned with the agile approach of continuous improvement. The notable downside is that there is rarely enough time to design a good API on top of an existing code base, because it might require a lot of refactoring. Thus the API might end up poorly designed exposing all the architectural design layers and missteps that the library went through in its early days.

The outside-in approach works in the opposite direction by first coming up with a good user interface for the problem in question. After that the implementation's job is to provide the functionality specified by the API. This approach usually creates nicer APIs that don't leak internal implementations as much, meaning the implementation can be changed more freely. Unfortunately this approach is typically more difficult, since it requires up-front knowledge about the problem space. Sometimes a predesigned API turns out to be unimplementable or require major changes.

In the real world most projects are a hybrid between these two approaches. In this chapter we are going to use the second approach, since we don't care what the actual API is, only what steps are needed to provide it.

15.2.1 The public header setup

The main entry point to any library is the header file it provides. Ducky's header is meant to be included like this:

```
#include<ducky/ducky.h>
```

The header has one struct definition and three function definitions, which together form the whole API. The struct definition is simple:


```
typedef struct _Ducky Ducky;
```

161

This declaration might be confusing for people who do not have earlier experience with C API declarations. C does not have classes, only structs. This defines the Ducky “object”, though it goes about it in a roundabout way.

In C if you have a struct `Foo` and you wish to define a variable of that type, you need to write `struct Foo varname`. Needing to write `struct Foo` all the time gets tiresome because what you’d really want to write is just `Foo varname`. This can be achieved with the `typedef` keyword, which creates new typenames.

The last piece of the puzzle is the declaration `_Ducky`, which is shorthand for saying that somewhere in the code there is a definition of a struct called `_Ducky`. Thus the declaration says the following: “define a new type called `Ducky` as an alias to a structure called `_Ducky`, whose definition is elsewhere”. This is a common C technique called an *opaque type declaration*. It affords us to define the API fully without specifying the layout of the Ducky object. In our case the Ducky object will be implemented in C++, so it could not even be defined in C.

With the data type defined we can now define the functions that make up the API.

```
Ducky* ducky_create(void);  
void ducky_squeeze(Ducky* d);  
void ducky_delete(Ducky* d);
```

The API follows the C convention of prefixing all function names with the library name to prevent symbol clashes.

15.2.2 Symbol visibility

As discussed in Chapter 2 every function and global variable creates a symbol. In order to use the functionality of a library, the user must be able to access those symbols. This gets problematic with shared libraries, because there are two different ways to handle this. Both of them are widely used so a portable solution must support both.

- In Microsoft’s Visual Studio toolchain, all symbols are hidden by default. Only those that are explicitly marked as exported will be available in the shared library.
- In most other toolchains *all* symbols are exported by default, even those that are only meant to be used internally.

Preventing the export of symbols that are only used in the implementation is important for three reasons. First of all it prevents users from accidentally

Table 15.1: Declarations for symbol visibility. Some Windows compilers also support the unixlike syntax.

	When compiling	When using
Windows	<code>__declspec(dllexport)</code>	<code>__declspec(dllimport)</code>
Unixlike	<code>__attribute__((visibility("default")))</code>	

calling into internal functions that they should not be using. The second is that it makes the exported symbol list smaller which speeds up linking. In our case we export only three symbols (the function names) but almost any program written in C++ creates hundreds, thousands, or in extreme cases tens of thousands of symbols for its internal use. The final advantage is that it produces slightly faster code, since the compiler can apply some optimisations to symbols it knows to be hidden that it can not apply to public symbols.

There is no clean and portable way of dealing with this problem. It must be solved with the C toolchains' biggest hammer: preprocessor macros. The public header provided by Ducky is used in two different ways: when building the library itself and when included from a different target that wants to use the library. The exported functions must be qualified with platform-specific declarations. The exact definitions are listed in Table 15.1.

Expanding all of that out to code we come up with the following macro forest that can be found at the top of the `ducky.h` header.

```
#pragma once
#if defined _WIN32 || defined __CYGWIN__
    #ifdef BUILDING_DUCKY
        #define DUCKY_PUBLIC __declspec(dllexport)
    #else
        #define DUCKY_PUBLIC __declspec(dllimport)
    #endif
#else
    #ifdef BUILDING_DUCKY
        #define DUCKY_PUBLIC __attribute__((visibility("default")))
    #else
        #define DUCKY_PUBLIC
    #endif
#endif
```

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct _Ducky Ducky;

DUCKY_PUBLIC Ducky* ducky_create(void);

void DUCKY_PUBLIC ducky_squeeze(Ducky* d);

void DUCKY_PUBLIC ducky_delete(Ducky* d);

#ifdef __cplusplus
}
#endif
```

After the exported functions get tagged with these declarations they look like the following:

```
DUCKY_PUBLIC Ducky* ducky_create(void);
void DUCKY_PUBLIC ducky_squeeze(Ducky* d);
void DUCKY_PUBLIC ducky_delete(Ducky* d);
```

The `Ducky` definition does not need to be tagged, because it does not generate any symbols, it is just a data type declaration.

This declaration works via the preprocessor symbol `BUILDING_DUCKY`. If it is set, then the header is being used when building the library and thus the `DUCKY_PUBLIC` definition gets expanded to the platform specific declaration to export the given symbol. If the symbol is not defined, then the header is being used to consume the library and `DUCKY_PUBLIC` gets expanded to a symbol import declaration. This requires that we add the definition `-DBUILDING_DUCKY=1` when building library sources. This will be handled later in this chapter.

15.3 Precompiled headers

C++ projects are notorious for their slow compile times. One main reason for this is that header files contain a major fraction of all functionality and they are parsed and instantiated for every source file that uses them. *Precompiled headers* are a tool designed to mitigate this problem by parsing the headers only once and storing them in a binary format that subsequent compilations can then use directly. Precompiled header files are also known as PCH files.

Meson has builtin support for using precompiled headers. The end user only needs to write the precompiled header file for each target and specify that it should be used when building a target. A precompiled header is nothing more than a regular header file that has `#include` declarations for all the headers that should be precompiled for the target. In this project we only use a PCH file for the main library and since the only external functionality we use is text printing, the PCH file is spartan:

```
#include<iostream>
```

Do **not** `#include` the PCH header file in any source file. Doing so can lead to breakages and even compilation slowdowns. Meson will take care of all the necessary magic behind the scenes. The setup is fairly complex and relies on the fact that users have not added any references to the PCH file in source files.

15.4 The C ↔ C++ bridge

The actual C++ implementation is simple, consisting only one class. Its name is `RubberDucky` to avoid confusion with the C side struct.

```
class RubberDucky final {
public:

    RubberDucky();
    ~RubberDucky();

    void squeeze();
};
```

The actual challenge comes when we need to make this C++ implementation appear to the outside world as if it was implemented in plain C. To achieve this we need to write some code that takes care of this domain translation. This is sometimes called a *bridge*, an implementation of the *adapter pattern*[7]. Let's start with the create function.

```
Ducky* ducky_create() {
    RubberDucky *r = new RubberDucky();
    return reinterpret_cast<Ducky*>(r);
}
```

Both lines in this function contain a construct that is discouraged in modern C++. Specifically using `new` directly rather than something like a `std::unique_ptr`

and the use of `reinterpret_cast`. This is typical for code at language boundaries.

The first line is fairly simple, as it only creates a new `Ducky` object. The second line is more interesting. It converts the `RubberDucky` pointer into a plain `Ducky` pointer (and thus to a `struct _Ducky` pointer), which is then returned to the caller. It may seem confusing that you can convert a C++ object to a C struct that is *not actually defined anywhere*. The reason this works is that even though the object types are completely different, *pointers* to them have the same underlying representation. The cast does not, in fact, produce any code. The pointer value returned by the `new` operator is returned directly.

This sort of code is *extremely* tricky and error prone. It is way too easy to accidentally trigger undefined behaviour with these operators leading to weird and hard to diagnose bugs. This case works because the only thing the caller of this function can do with the return value is pass it back to this library as an argument. Then we can cast it back like this.

```
void ducky_squeeze(Ducky* d) {
    RubberDucky *r = reinterpret_cast<RubberDucky*>(d);
    r->squeeze();
}
```

This cast is the inverse operation of the cast in the constructor. Once we have a C++ object pointer we can call its member functions. The delete method's implementation is almost identical.

```
void ducky_delete(Ducky* d) {
    RubberDucky *r = reinterpret_cast<RubberDucky*>(d);
    delete r;
}
```

Even for this simple example the language interop code requires expert level domain knowledge. Therefore it is always a good idea to keep this code in its own file away from the actual “business logic” code.

15.5 Tests

Since the library provides so little functionality, writing tests for it is simple.

```
#include<ducky/ducky.h>

int main(int argc, char **argv) {
```

166

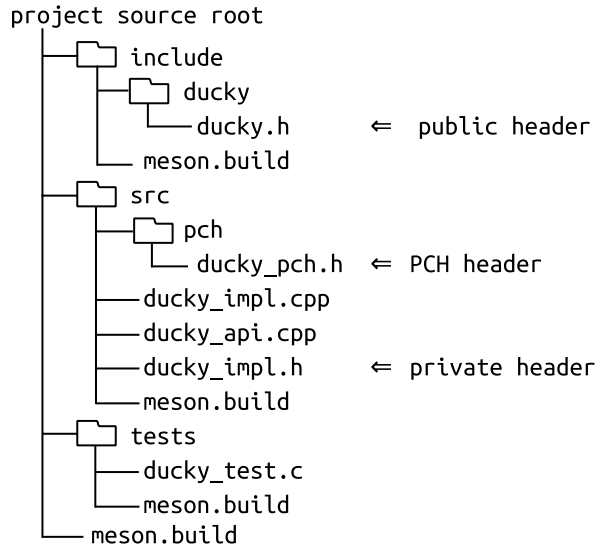


Figure 15.1: File layout of sample library project. The private header must not be exposed to users of this library.

```

Ducky *d = ducky_create();
ducky_squeeze(d);
ducky_delete(d);
return 0;
}

```

There is one catch, though. Since our library exposes a C API, we must test that the library can be used from plain C. Because this the test executable is written in C and not C++.

15.6 Project layout

The layout of the project with all files and directories can be seen in Figure 15.1. At this point we have all the information necessary to write out the final build definition files.

15.6.1 The include directory

The build definition for the `include` directory is the simplest of them all. It consists of only one directive.

```
install_subdir('ducky', install_dir: get_option('includedir'))
```

We use the `install_subdir` function to install the `ducky` directory and all its contents to the global header directory. In practice this installs the project's one public header file to `/usr/include/ducky/ducky.h`.

15.6.2 The source directory

Building the actual library gets more difficult, but not by much. There is still only one Meson function invocation.

```
ducky_lib = library('ducky',
    'ducky_impl.cpp', 'ducky_api.cpp',
    include_directories: ducky_inc,
    install:             true,
    cpp_args:             '-DBUILDING_DUCKY=1',
    cpp_pch:              'pch/ducky_pch.h',
    gnu_symbol_visibility: 'inlineshidden',
)
```

We use the `library` function rather than, say, `shared_library` in order to allow the end user to decide whether to build a shared or static library. The library name and source definitions follow the standard values on each platform.

Building the library requires access to the public headers, which come via the `ducky_inc` variable. We also set `install` to `true` so the resulting library gets installed.

The precompiled header to use is specified with the `cpp_pch` kwarg. The header itself is in a separate `pch` subdirectory. Meson requires you to put the `pch` file in a separate subdirectory as it is a requirement to make precompiled headers work reliably on all platforms and toolchains. Do *not*, under any circumstances, add the `pch` directory to any target's include directories. Meson will take care of all include path manipulation for PCH files.

As was discussed in Section 15.2.2, we need to set a compiler define when building the library so the macro definitions can do their magic. This is set with the `cpp_args` keyword argument. The last keyword argument is for GCC and related compilers. They make all symbols public by default. Since we don't want that we set all symbols to be hidden with `inlineshidden`. The difference

between that and plain `hidden` have to do with how implicitly defined functions in C++ classes get handled. Since we only export a plain C API we *don't* want those exported so we pick the stronger value.

15.6.3 The tests directory

The definition for tests is almost as simple as it was for the include directory.

```
test_exe = executable('ducky_test', 'ducky_test.c',
    dependencies: ducky_dep)
test('ducky_test', test_exe)
```

First we build a test executable and then define a test that runs it. The only unexpected piece is the dependency object `ducky_dep`. That is defined in the top level `meson.build` file, which is what we'll examine next.

15.6.4 The root directory

This is the final file and where the entire build definition comes together. It is the longest and most complex file of all, but it should still be fairly readable and understandable.

```
project('ducky', 'c', 'cpp',
    version: '0.1',
    default_options: ['cpp_std=c++14'])

subdir('include')
# Needed to build the library.
ducky_inc = include_directories('include')
subdir('src')

# Needed to be usable as a subproject.
ducky_dep = declare_dependency(
    include_directories: ducky_inc,
    link_with: ducky_lib)

subdir('tests')

# Needed to be discoverable when installed.
pkg_mod = import('pkgconfig')
pkg_mod.generate(
    name: 'Ducky',
```



```
    filebase:    'ducky',  
    description: 'You are never alone with a rubber duck.',  
    libraries:   ducky_lib,  
  )
```

The definition starts with the standard project declaration and then recurses in the `include` subdirectory so the header installation declaration gets processed.

Then it creates an `include_directories` object for the header directory which is needed in all compilation jobs in this project. Then we descend in the `src` subdirectory where the library gets built.

One of the main requirements we had was that this project must be usable as a Meson subproject. Most of that work is done in the next `declare_dependency` function call. It creates a dependency object that anyone can use to build against this library. This definition consists only of the include directory and one library to link against. This object is also used to build the test executable as mentioned earlier. In this way we also verify that the dependency object is properly usable. If it was broken, the test executable would not build.

The last piece is the `pkg-config` declaration. Most things in it are self-explanatory except perhaps the `filebase` keyword argument. It is used to define the name of the installed `pkg-config` file, which specifies its `pkg-config` module-name. In this case the filename will be `ducky.pc`. The recommended practice is that the module's name should only consist of lower case letters, dashes and numbers and it should be the name of your project in lower case if possible.

15.7 Creating releases

The project is now buildable and tested to work. The next step is to create a release that can be given to other people to use. Some people do this merely by adding a tag in their source repository and have people use that. The other approach is to create a distribution archive, also known as a *release tarball*, which contains the same files as the tagged version, but no revision control history or metadata. This makes it small and self contained.

Release archives are required for many policy and legal requirements so it is considered good form to provide them for every release. Merely putting all source files in a Zip archive is not sufficient, though, since there are specific naming requirements. The archive file name must be `projectname-version.tar.xz`, which is a tar file compressed with the LZMA compression algorithm. Until a few years ago most archives used the Zlib compression instead and had the file extension `.gz`. Since then there has been a shift to the LZMA format as it provides noticeably better compression ratios. Inside the archive all files must be placed in a directory called `projname-version`.

Meson provides the `dist` command that automatically creates a release archive in the current form.

170

```
$ ninja dist
[0/1] Creating source packages
Cloning into '<<builddir>>/meson-dist/ducky-0.1'...
done.
Testing distribution package
  <<builddir>>/meson-dist/ducky-0.1.tar.xz
The Meson build system
Version: 0.50.999
Source dir: /tmp/tmpqj2pnckg/ducky-0.1
Build dir: /tmp/tmpyp2xv04p
Build type: native build
Project name: ducky
Project version: 0.1
Native C compiler: cc (gcc 8.2.0)
Native C++ compiler: c++ (gcc 8.2.0)
Build machine cpu family: x86_64
Build machine cpu: x86_64
Build targets in project: 2
Found ninja-1.8.2 at /usr/bin/ninja
[7/7] Linking target tests/ducky_test.
[0/1] Running all tests.
1/1 ducky_test                                OK                0.00 s

Ok:                                1
Expected Fail:                     0
Fail:                              0
Unexpected Pass:                   0
Skipped:                           0
Timeout:                           0

Full log written to /tmp/tmpyp2xv04p/meson-logs/testlog.txt
[0/1] Installing files.
Installing subdir /tmp/tmpqj2pnckg/ducky-0.1/include/ducky to
/tmp/tmp89rf6h/usr/local/include/ducky
Installing /tmp/tmpqj2pnckg/ducky-0.1/include/ducky/ducky.h to
/tmp/tmp89rf6h/usr/local/include/ducky
Installing src/libducky.so to
/tmp/tmp89rf6h/usr/local/lib/x86_64-linux-gnu
```

```
Installing /tmp/tmpyp2xv04p/meson-private/ducky.pc to
/tmp/tmp89rf6h/usr/local/lib/x86_64-linux-gnu/pkgconfig
Distribution package <<builddir>>/meson-dist/ducky-0.1.tar.xz
tested.
```

171

This run shows that in addition to creating the archive, Meson also verifies that it works by extracting, compiling, running the full test suite followed by running install. Only if all these steps are successful does it create the output files in `meson-dist` directory.

```
$ ls meson-dist
ducky-0.1.tar.xz  ducky-0.1.tar.xz.sha256sum
```

In addition to the archive there is a checksum file in the format required by some upload processes. An optional, but strongly recommended, step is to cryptographically sign the output archive with a tool such as GPG. This way users of your software can reliably verify that they have downloaded an authentic release rather than a malicious one. The release is now ready to be shared with the world.

15.8 Exercises for the reader

This library setup is simple and can be expanded in many ways. Here are some possible exercises for the reader in increasing order of difficulty.

1. Verify that the library also works when used from C++ by adding a new test executable.
2. Generate the header file from a template file with `configure_file` so that it contains the project's version number as a preprocessor define.
3. Create a version 2 of the Ducky library so that both versions are parallel installable on the same system.
4. Expand the above so that both versions of the library can live inside the same process.

Chapter 16

Practical tips for real world projects

Whenever you start using a new piece of technology, you eventually find individual usage tricks and processes that are generally useful but are not strictly classifiable. That is where this chapter comes in. It lists various real world use cases, some things to try and, just as importantly, things not to do.

The information in this chapter is probably most useful only after you have personal usage experience with Meson. First timers might consider reading this chapter again after having used Meson in their projects for a few weeks.

16.1 Use options rather than hardcoding compiler flags

Let's start this chapter by looking at the most common antipattern for new Meson users. Many developers have gotten used to configuring their builds by manually setting command line arguments. The most common ones include `-O2`, `-g`, `-Wall` and `-pipe`. These can be injected many different ways including environment variables when first configuring the build dir, `add_project_arguments` and cross and native files.

None of these should be used. They are not supported by all compilers and are thus nonportable. In addition they clash with Meson's builtin options for these, which are `optimization`, `debug` and `warning_level`, respectively. Meson will add `-pipe` automatically when the underlying compiler supports it¹. An additional bonus is that you can change these values on the command line via `meson configure`. Specifying these in build files means having to edit them to make any changes.

¹The `-pipe` flag did improve compilation speeds in the 90s, but it is arguable whether it provides performance benefits any more and should be dropped altogether.

16.2 Shipping pregenerated files

174

The recommended workflow for Meson is to generate all necessary files on demand every time rather than storing and using pregenerated intermediate files. Sometimes this is cumbersome, though, especially if you need a lot of heavy third party dependencies to generate the output files. In practice this seems to be most common for man pages. To make things easier for users, you'd want to enable the following work flow:

- During development, build man pages from sources.
- When generating a release tarball, add the generated man pages in the release.
- When building the final release, install the embedded man pages rather than generating them.

This can be achieved with the following build definition file.

```
project('manpregen', 'c',
        version: '1.0.0')

fs = import('fs')
mandir = get_option('mandir') / 'man1'

if fs.exists('program.1')
    message('Installing pregenerated manpage.')
    install_data('program.1',
                 install_dir: mandir)
else
    # Only try to look up external dependencies
    # inside this block, when we know that we actually
    # need them.
    gen = find_program('heavy_program.py')
    message('Building manpage from scratch.')
    custom_target('manpage',
                  output: 'program.1',
                  command: [gen, '@OUTPUT@'],
                  install: true,
                  install_dir: mandir)
endif
```

```
meson.add_dist_script('dist_script.py')
```

175

The implementation uses the `fs` module, whose documentation can be found in Section 20.3. The actual logic is readily understandable. If a pregenerated man page can be found in the current source directory, then install it directly, otherwise generate it from sources as usual. The only remaining problem is how to get the pregenerated file inside the release archive. This is done by using the `meson` object to add a dist script. See Section 18.12.1 for details.

Its contents are in all simplicity the following:

```
#!/usr/bin/env python3

import os, shutil

gen_file = os.path.join(os.environ['MESON_BUILD_ROOT'],
                        'program.1')
shutil.copy(gen_file, os.environ['MESON_DIST_ROOT'])
```

All it does is use the environment variables set by Meson to copy the generated file from the build directory to the staging directory, which will be turned into the final release archive. When the archive is built, the log output shows it is working as expected:

```
$ meson build
The Meson build system
Version: 0.54.999
Build type: native build
Project name: manpregen
Project version: 1.0.0
C compiler for the host machine: cc
(gcc 9.3.0 "cc (Ubuntu 9.3.0-10ubuntu2) 9.3.0")
C linker for the host machine: cc ld.bfd 2.34
Host machine cpu family: x86_64
Host machine cpu: x86_64
Message: Installing pregenerated manpage.
Build targets in project: 1
Found ninja-1.10.0 at /usr/bin/ninja
```

16.3 Do not treat files as strings

176

Many people coming from other build systems, especially those used to writing plain Makefiles, often think about the build in terms of strings and command line arguments. While one can write working Meson files in this way, it is not recommended. As an extreme example, several people have written build files like these when first porting their projects to Meson.

```
# Bad example, do not do this!
sources = [meson.current_source_dir() / 'file1.c',
           meson.current_source_dir() / 'file2.c',
           meson.current_source_dir() / 'file3.c',
]

# In a different directory.
executable('program', sources)
```

This way of writing files is unreadable and requires a lot of unnecessary work. The intent of the author is clear in that they want to define sources in one directory and use them in a different one. In string-based build systems building full paths like this is the only feasible way to do it. The “mesonic” way of doing this is to use the `files` function.

```
sources = files('file1.c', 'file2.c', 'file3.c')

# In a different directory.
executable('program', sources)
```

Using `files` provides three main advantages. First, it is more readable. Second, it validates that all files referred by the arguments exist. Third, the result object can be used in any other directory and Meson will automatically expand the paths to point in the correct place.

Another thing people coming from Make-based systems often seem to do is to build command line arguments by hand. A typical case might go like this:

```
# Bad code, do not do this.
sources1 = ['input1', 'input2', ...]
sources2 = ['inputa', 'inputb', ...]

custom_target('custom',
              output: 'output.dat',
              command: ['script',
```



```
meson.current_source_dir(),
meson.current_build_dir() / 'output.dat']
+ sources1 + sources2)
```

177

The generator script would then take these arguments, generate the input files manually and write the output file. Just like above this is not very readable, but, even worse, is not reliable either. Because the arguments are strings, Meson has no way of knowing what the input files are and thus can not set up dependencies properly. If you change one of the input files, the output file is not regenerated. This is a serious error and the build system has no way of getting it right. The target should instead be defined like this:

```
sources1 = files('input1', 'input2', ...)
sources2 = files('inputa', 'inputb', ...)

custom_target('custom',
  output: 'output.dat',
  input: sources1 + sources2,
  command: ['script',
            '@OUTPUT@',
            '@INPUT@']
)
```

Now Meson knows all the inputs and outputs and can set things up so that the output file is generated if and only if any of its input files change.

16.4 Running Python scripts that use extension modules

Most scripts that are run in builds are self contained and can easily be run directly. Things get more complicated when they start requiring external dependencies. If they come from the system things will still work out of the box (assuming the dependencies are installed), but things are not as simple if the dependencies come from your own project. In this section we'll look at how to make Python module imports work, but the same general approach works with almost all other scripting languages.

The common case is that you have a script such as `test/script.py` and it imports a module in the directory `pymod/depmod`. If you try to run the script directly, it will fail with a module import error that looks something like this:

```
Traceback (most recent call last):
  File "/path/to/script.py", line 5, in <module>
```

```
import depmod
ModuleNotFoundError: No module named depmod'
```

There are three different ways of solving this problem, all with their own pros and cons. The arguably simplest solution is to keep all your Python code in a single directory. This makes Python treat them all as part of the same unit and they can import functionality from each other directly. In this example this would be done by moving `script.py` inside the `pymod` directory. The downside is that scripts are no longer next to the place there are used in, so you can't use them with a simple `find_program('script.py')` but instead have to write a relative path to the module dir.

The second approach is to alter the module import directories from within the script at runtime. The exact details can be found in Python's documentation, but roughly you would do something like this:

```
import sys
my_mod_dir = calculate_path_to_moddir()
sys.path.append(my_mod_dir)

# Rest of the code goes here.
```

The main downside of this approach is that you should never use it in scripts that get installed via, e.g. the Python Package Index (or PyPi). System dependencies that change the module lookup path are highly frowned upon. This is not a problem with internal scripts, such as test programs.

The third solution is to set the environment variable `PYTHONPATH` to point to your internal Python modules. Here's how it would be done for a unit test.

```
test('pythontest', find_program('script.py',
    env: 'PYTHONPATH=' + meson.current_source_dir() / '../pymod')
```

If `PYTHONPATH` is needed in many different places, you'd probably want to set it to a variable in the main `meson.build` file so it is available everywhere. This approach could be considered "the most correct" but it has one major downside: the script can no longer be run directly from the command line. You have to always run it via Meson or set up `PYTHONPATH` by hand, which is a bit tedious. If these limitations are not an issue in your use case, then this is probably the approach you should go for.

16.5 Move everything you can out of build files

179

The last piece of advice in this book is going to be a bit counterintuitive. After more than 150 pages of going through all the things you can do with Meson build files, we are now going to look at how and when to *not* use them. The basic rule for this is simple: whenever you can move something away from the build definition file, you probably should do that. The build file should be as much as possible about how your code gets built and as little as possible about how the build needs to be adapted to the environment.

In many other build systems, these two things are intermixed. By contrast Meson aims to provide builtin tools and features to keep these two concepts isolated from each other. Say you have an embedded project that needs to build on two different hardware platforms that require different compiler flags. A “traditional” build system approach would lead to a build file looking like this:

```
# Bad code, do not do this.
if building_for_platform_one
    add_global_arguments('-DSOME_PLATFORM',
                        language: 'c')
else
    add_global_arguments('-DOTHER_PLATFORM',
                        language: 'c')
endif
```

This works but is not very elegant. The more features like these you have the more convoluted your build file becomes, eventually turning it into a mess. The better way to achieve the same result is to move these flags to the cross file’s [properties] section:

```
[properties]
c_args = ['-DSOME_PLATFORM']
```

Not only does this declarative approach completely get rid of `if/else` spaghetti but it has other benefits. Compiler arguments specified in the build file are not used in things like compilation checks, whereas those defined in the cross file are used. This is almost always what you want and this approach makes it work out of the box. The separation is even more important for code that is designed to be built as a subproject, especially on platforms not tested by the original developer. A good rule of thumb to aim for is that people should be able to build the project on new platforms without needing to edit the build definitions. Sometimes it is not possible, but when it is it makes it a lot easier for other people to consume your code.

There are many other places where this approach should also be taken. Perhaps the weirdest is that some projects want to reinvent Meson's builtin functionality (as we saw in Section 16.1) for things such as debug info, sanitizers and other things. This code is not only superfluous, but actively harmful because they are confusing to new contributors, because their existing knowledge is not transferable and they will try to use the builtin functionality. This will, at best, maybe work and at worst will cause the two approaches to clash, burn and lead to incomprehensible errors and a system that is broken but no-one can really tell why or how it should be fixed. Fortunately the solution is simple: delete all these custom things from your build files. The end result is less code and a system that works more reliably.

Part II:

The reference documentation

Chapter 17

Elementary object reference

The remaining chapters contain detailed description of Meson's functionality. The bulk of it consists of functions and methods. They are formatted in the following way:

```
sample_function(args, ...) → return type
```

There are four pieces of information on each declaration. The first is the name of the function or module, in this case `sample_function`. This is followed by positional arguments, which are listed in the command, and keyword arguments which are not. Instead any function that takes keyword arguments contains an ellipsis (...) as its last argument and the actual arguments are documented later in the section. Trying to cram keyword arguments on the same line would take too much space and get quite messy. The last piece of information is the return value of this function or `none` if nothing is returned. Most functions have some sample code showing how they can be used. This is omitted for simple functions that, for example, only return a specific string.

This chapter documents *elementary objects* such as integers, strings, files and arrays. These are types that you could expect to find in most programming languages. In other words they are not *domain-specific* types, which are documented in the next chapter.

17.1 array object

An array is an immutable sequence of arbitrary objects, including other arrays. Note that arrays can not be cyclical, that is, an array can not contain itself as an element, even indirectly.

Arrays can be concatenated with the `+` operator.

184

```
new_array = ['a'] + ['b']
# new_array is ['a', 'b']
```

Individual elements of an array can be obtained by indexing or with the `get` method described in Section 17.1.2. Negative indexes count backwards from the end of the array. Trying to access an out of bounds object is a hard error.

```
x = ['a', 'b', 'c']
x[0]           # Result is 'a'
x[-1]          # Result is 'c'
x[4]           # Index out of bounds => hard error
x[-4]          # Index out of bounds => hard error
```

Since arrays can not be altered, the way to extend an array is to create a new array and assign it to the same variable. Since this is such a common operation, Meson provides the `+=` operator as a shorthand.

```
x = ['a']
x += ['b']
# x is ['a', 'b']
```

This is equivalent to the following code:

```
x = ['a']
x = x + ['b']
```

If `x` had been used the old value would remain in effect in all the places it was used in. Changing `x` like this only affects uses of `x` from this point on.

17.1.1 `contains(object)` → boolean

Returns `true` if the array contains the object given as the argument.

Positional arguments

object The object to search for.

Usage examples

```
x = ['a', 1]
x.contains('c') # returns false
```



```
x.contains('a') # returns true
```

185

17.1.2 `get(index, default_value) → object`

This method behaves the same as indexing an array, but allows specifying a default value in case the index is invalid.

Positional arguments

index An integer for the index that should be accessed

default_value An optional value that is returned if the index is invalid, i.e. out of bounds.

Usage examples

```
arr = ['a', 'b', 'c']
arr.get(0)           # Result is 'a', identical to arr[0]
arr.get(4)           # Hard error.
arr.get(4, 'default') # Result is 'default'
```

17.1.3 `length() → integer`

Returns the number of elements in this array. The count is not recursive, that is, if any item in the array is itself an array, it is counted as one item regardless of how many items it contains.

Usage examples

```
[].length()           # Result is 0
['a'].length()         # Result is 1
[['a', 'b', 'c']].length() # Result is 1
```

17.2 boolean object

A boolean object is either the value `true` or `false`.

17.2.1 `to_int() → integer`

Returns 1 if the boolean was `true` and 0 if it was `false`

17.2.2 to_string(true_string, false_string) → string

This method can be used to obtain a string representation of the boolean. The default representations are the strings 'true' or 'false'.

Positional arguments

true_string The string that should be returned for **true**.

false_string The string that should be returned for **false**. Either both or neither of these arguments must be defined. Having only one is a hard error.

Usage examples

```

true.to_string()           # Returns 'true'
true.to_string('yes', 'no') # Returns 'yes'
false.to_string('yes', 'no') # Returns 'no'

```

17.3 dictionary object

A dictionary is a set of unique strings called *keys* each of which maps to a specific *value*. This data type is also known as the *associative array* or a *hash map*. Dictionaries are used in the same way as arrays.

```

d = {'key1': 'value1',
     'key2': 'value2'}
d['key1']           # Result is 'value1'
d['nonexisting']    # Hard error.

```

Just like arrays, dictionaries can be combined with + and +=. When two dictionaries are added and both of them have the same key, the result object will take the value from the latter.

```

d1 = {'key1': 'value1'}
d2 = d1 + {'key2': 'value2'} # d2 has two keys
d3 = d1 + {'key1': 'new'}    # d3 has one key with value "new".
d3 += {'key3': 'value3'}     # d3 has two keys

```

17.3.1 has_key(keyname) → boolean

This method tells whether a given string is used in the dictionary as a key (but not value).

Positional arguments

keyname A string argument listing the key to be checked.

187

Usage examples

```
d = {'key', 'value'}
d.has_key('key')      # returns true
d.has_key('balue')    # returns false
```

17.3.2 get(key, default_value) → object

Obtains the value for the given key with an optional value to return if the key is not found.

Positional arguments

key is the key to look up.

default_value is an optional value to be returned if the key does not exist in the dictionary.

Usage examples

```
d = {'key', 'value'}
d.get('key')           # Returns 'value'
d.get('key', 'fallback') # Returns 'value'
d.get('bob')           # Hard error
d.get('bob', 'fallback') # Returns 'fallback'
```

17.4 disabler object

The disabler object is perhaps Meson's most peculiar object type. It does only one thing: if a disabler object is used as an argument for any operation it will cause the operation to not be run and instead return a disabler object. This can be used to disable parts of the build easily without needing to write a lot of branching code.

```
d = disabler()
x = 1 + d                    # x is disabler
e = executable(..., dependencies: d) # e is disabler
```

```
if d
    something()          # This branch is not taken.
else
    something_else()     # And neither is this.
endif
```

17.4.1 `found()` \rightarrow boolean

This is the only operation you can do with a disabler object that does not short circuit a return value of disabler. This allows disablers to be used like dependency objects.

Usage examples

```
d = disabler()
d.found() # Returns false.
```

17.5 integer object

An integer object is an integer, that can be used in standard mathematical operations using `+`, `-`, `*` and `/`. Attempting to divide by zero is illegal, and will cause an unrecoverable error.

17.5.1 `is_even()` \rightarrow boolean

This method returns `true` if the integer is even, that is, its remainder when divided by 2 is zero.

Usage examples

```
1.is_even() # False
-1.is_even() # False
2.is_even() # True
-2.is_even() # True
0.is_even() # True
```

17.5.2 `is_odd()` \rightarrow boolean

This function is the opposite of `is_even`.

Usage examples

```
1.is_odd()    # True
-1.is_odd()   # True
2.is_odd()    # False
-2.is_odd()   # False
0.is_odd()    # False
```

189

17.5.3 More elaborate usage examples for integers

```
x = 1 + 2 * 3    # x has the value 7.
y = (1+2) * 3    # y has the value 9.
bad = 7/0        # Meson will exit with a hard error.
```

17.6 string object

A string contains zero or more UTF-8 encoded characters and is immutable. It is not possible to express arbitrary binary data with Meson strings. The only place where this may become an issue is that some operating systems permit filenames that can not be expressed with Unicode. Files with these names can not be expressed in Meson and thus they can not be used in build targets. All string operations are case sensitive.

The simplest way to combine strings is the + operator:

```
food = 'ham' + 'burger' # Result is 'hamburger'.
```

You can also join strings as if they were file system path segments by using the / operator. If the element on the right side of the division operator is an absolute path, the element of the left side is discarded.

```
seg1 = 'sub1'
seg2 = 'sub2/sub3'
abs = '/root'

seg1 / seg2      # Result is 'sub1/sub2/sub3'
abs / seg1       # Result is '/root/sub1'
abs / seg1 / seg2 # Result is '/root/sub1/sub2/sub3'
seg1 / abs       # Result is '/root'
```

17.6.1 `contains(item)` → `boolean`

Returns `true` if the string contains the given argument string as a substring.

Positional arguments

item is the string that should be searched for.

Usage examples

```
'team'.contains('I') # False, there is no "I" in "team"
'abcd'.contains('cd') # True
```

17.6.2 `endswith(suffix)` → `boolean`

Returns `true` if the string ends with the string given as the first argument.

Positional arguments

suffix The string suffix to be checked.

Usage examples

```
'hello'.endswith('hi') # False
'hello'.endswith('lo') # True
```

17.6.3 `format(format_arguments)` → `string`

This method is the most versatile way of producing new strings. It works by going through the string looking for markers of type `@number@` and replacing those with the corresponding positional arguments. Indexing is zero-based. If any entries point to entries that do not exist, such as trying to access the fourth entry but only three arguments have been given, Meson will exit with a hard error.

Positional arguments

format_arguments The arguments are the values to write to the result string. They can be any object that has a natural string representation such as a string, integer or boolean. Meson will take care of the conversion for you. If you need full control over the conversion, then you need to construct the

substitution string manually and pass the result string as the argument to this method.

191

Usage examples

```
template = '@0@ is @1@.'
res = template.format('Thirteen', 13)
# res contains the text 'Thirteen is 13.'
```

17.6.4 `join(strings_to_join) → string`

The `join` method creates a string by joining the given argument strings. The string being used to call the method is inserted between each string segment.

Positional arguments

`strings_to_join` The argument list consists of the individual strings that will be concatenated, in order, to the final string.

Usage examples

```
x = ', '.join('a', 'b', 'c', 'd')
# x has the value 'a, b, c, d'
y = ''.join('a', 'b', 'c')
# y has the value 'abc'
```

17.6.5 `split(split_segment) → string array`

This method is the opposite of `join`, it splits a string on every instance of the given split segment.

Positional arguments

`split_segment` is an options separator string that should be used to do the splitting. If no specific split character is specified, the string is split on whitespace characters.

Usage examples

```
x = 'one two three'.split()
# x is ['one', 'two', 'three']
```

```
y = '127.0.0.1'.split('.')
# y is ['127', '0', '0', '1']
```

17.6.6 `startswith(prefix)` → boolean

This method is conceptually the same as `endswith` except that it checks whether the string ends with the given string.

Positional arguments

prefix The string value to be checked.

Usage examples

```
'prefixinator'.startswith('suffix') # False
'prefixinator'.startswith('prefix') # True
```

17.6.7 `strip(strip_characters)` → string

This method is used to clear unneeded characters from both ends of a given string. The most common use case is to remove whitespace characters such as linefeeds.

Positional arguments

strip_characters An optional string containing all the characters that should be removed. If not specified the function will remove whitespace characters, that is, spaces, tabs, linefeeds and carriage returns.

Usage examples

```
s1 = ' text \n'.strip()      # s1 is 'text'
s2 = ' text '.strip('tex')   # s2 is ' text ' (nothing is removed)
s3 = 'atexta'.strip('a')     # s3 is 'text'
s4 = 'atextb'.strip('a')     # s4 is 'textb'
s5 = 'atextb'.strip('ab')    # s5 is 'text'
```

17.6.8 `to_int()` → integer

Converts the string to an integer in base 10. If the string does not represent a number, the result will be a hard error.

Usage examples

```
one = '1'.to_int()
minus_one = '-1'.to_int()
hard_error = 'text'.to_int() # Terminates the program.
```

193

17.6.9 to_lower() → string

This method, unsurprisingly, converts a string to all lower case.

Usage examples

```
l1 = 'o0o'.to_lower() # l1 is 'ooo'
l2 = 'o0o'.to_lower() # l2 is 'ooo' (middle character is zero)
```

17.6.10 to_upper() → string

This method converts the string to all upper case.

Usage examples

```
l1 = 'abc'.to_upper() # l1 is 'ABC'
l2 = 'a_a'.to_upper() # l2 is 'A_A'
```

17.6.11 underscoreify() → string

This helper method can be used to convert strings to a format that are usable as variables in most programming languages, especially those of the C family. This is done by replacing all characters that are neither latin characters nor numbers with the underscore (`_`) character.

Usage examples

```
u1 = 'parted-name'.underscoreify() # u1 is 'parted_name'
u2 = 'directory/path'.underscoreify() # u2 is 'directory_path'
u3 = 'a!b"c@d%f&'.underscoreify() # u3 is 'a_b_c_d_f_'
```

17.6.12 `version_compare(comparison_string) → boolean`

194

The simplest way to compare two strings is to go through both strings one character at a time and compare each pair in isolation. This is the typical way of sorting strings and can be used to easily detect that, for example, the string 'abc' is less than (meaning should be sorted before) the string 'def'.

This comparison does not work when dealing with version numbers. Suppose we have two version strings '10' and '3'. A simple comparison starts by comparing the first elements of each string. It would note that '1' is less than '3' and would sort '10' before '3'. This is semantically incorrect, since the number 3 is less than 10 and should order before it.

This method provides a way to compare version strings in a “natural order”, which treats digit sequences in the string as numbers rather than individual characters. More specifically this method compares version strings using the *semantic versioning* schema of period separated numbers. The most common format is `major.minor.bugfix-extra`, for example `1.0.0`. There can be more or fewer number elements, but three is the most common number. The `-extra` part can be any string, such as `-rc1` for release candidate one. There are no limitations on what the extra part can contain, so it is recommended that you set up your version number checks so that its contents do not affect the results.

Positional arguments

comparison_string A string that defines the version check. It consists of two parts. The first one is the comparison specification, and it can be any of `<`, `<=`, `>`, `>=`, `==` or `!=`. This should be immediately followed by the version number.

Usage examples

```
v = '1.2.3'
v.version_compare('<2.0.0') # true
v.version_compare('<=1.2.3') # true
v.version_compare('<1.10.3') # false
v.version_compare('!=3')    # true
v.version_compare('>1.2')    # true
```

Chapter 18

Domain specific object reference

This chapter documents all objects that define parts of the build, such as executables and other build targets. This includes objects used for constructing custom build steps.

18.1 `build_machine` object

The build machine object provides information about the computer system being used to compile the current program. See Chapter 12 for an explanation about the different machines and their usage.

The build machine object is always set and can not be changed. Trying to assign a value to the `build_machine` variable is a hard error.

18.1.1 `cpu_family()` → `string`

Some computer processors have many different makes and models that are compatible, but are known by different names. The most well known example of this are 32 bit Intel x86 processors. Over the 30+ year history the processors have had a plethora of names, both release and code names all of which are used *somewhere*. Sometimes different operating systems call the same processor with different names. Thus a x86 processor might be know as *x86*, *i586*, *'pentium'* or some other value

For most configuration purposes the exact make and model of the processor does not matter, only the “family”, that is, a common name for all these processor types. This function provides this name, so that all build scripts don’t have to reinvent the logic matching processor names to a generic family name.

The most common return types are `x86` and `x86_64` for 32 and 64 bit Intel and AMD processors and `arm` and `aarch64` for 32 and 64 bit ARM processors. New processor families are added to Meson all the time. A full and up to date list of supported CPU families can be found on the Meson project's home page.

Usage examples

```
# on a 64 bit Intel or AMD machine
build_machine.cpu_family()    # Result is 'x86_64'.
```

18.1.2 `cpu()` → `string`

Returns a string describing the processor on the build machine as accurately as possible. There are no set form for the string. Different operating systems may return a different value when run on the same machine.

Usage examples

```
# On a 32 bit ARM machine
build_machine.cpu()
# Result may be 'arm', 'arm5', 'arm7hle' or some other
# value completely.
```

18.1.3 `system()` → `string`

Returns a string describing the build machine's operating system. This value is derived mostly from the operating system kernel. For example Apple has devices that run both macOS and iOS, but both of them report the system as *darwin*. On the other hand the Cygwin environment runs directly on top of Windows, but this value is *cygwin*. This may seem a bit strange but it is done to maintain consistency with other tools that have a similar naming scheme.

The most common system names are `linux`, `windows` for all versions of Windows, and `darwin` which is used both for macOS as well as iOS. Support for new systems is added to Meson fairly regularly. A complete and up-to-date list of supported systems can be found on the Meson project's home page.

Usage examples

```
# On any version of Windows.
build_machine.system()    # Result is 'windows'.
```

18.1.4 `endian()` → `string`

Returns the processor's endianness, which is either `big` or `little`.

197

Usage examples

```
build_machine.endian() # 'little' on most modern processors
```

18.2 `build_target` object

A build target object represents the output of all targets that compile source code to binary targets. In practice this means all targets except `custom_target` and `run_target`.

18.2.1 `extract_all_objects()` → `object`

Returns an opaque object that holds all the object files that get built in this target. These can be passed to other build targets using their `object` keyword argument, which will cause the object files to be linked in the other target directly. Note that this should only be used for special cases. Most of the time you should build a library and link that to the target.

Keyword arguments

`recursive` specifies whether the result should also contain those object files that were passed in to the target rather than just the ones that are built as part of the target. Defaults to `false`.

18.2.2 `extract_objects(sources, ...)` → `object`

Like `extract_all_objects`, except that it only extracts the objects files of the sources given as arguments.

Positional arguments

`sources` are one or more input sources that can be either strings or file objects.

Usage examples

```
shlib = shared_library('name',  
                        'source1.c',
```

```
        'source2.c')
single_obj = shlib.extract_objects('source2.c')
```

198

18.2.3 full_path() → string

Returns the absolute path to the output file. Note that usually you don't need this, using the object directly in other targets makes Meson expand the path automatically and set up proper dependencies between the two targets.

18.2.4 name() → string

Returns the name of the target as a string.

18.2.5 private_dir_include() → include directory object

Returns an include directory object that points to the target's private directory. This might be needed in cases where you generate headers in the private dir, but you also need the headers in other compilations.

18.3 compiler object

A compiler object represents a compiler for a single language. Compiler objects can be obtained from the `meson` object's `get_compiler` method. The two most common operations are getting the compiler's name for configuration purposes and running compilation checks which are available as methods. Not all languages support all the checks. These checks are most commonly needed with the C family of languages.

All compile test checks accept the same set of keyword arguments. Rather than listing them in every method description, they are listed collectively in Section 18.3.28.

18.3.1 alignment(type, ...) → integer

Returns the alignment of the specified type.

Positional arguments

type the type to be checked.

Usage examples

```
# Ints usually have an alignment of 4.
a = cc.alignment('int')

# System structs can be inspected too.
a = cc.alignment('struct stat',
                 prefix: '#include<sys/stat.h>')
```

199

18.3.2 `command_array()` → string array

Returns an array of strings specifying the compiler command.

Usage examples

```
# A typical answer might be ['ccache', 'gcc']
carr = cc.command_array()
```

18.3.3 `check_header(name, ...)` → boolean

Returns `true` if the specified header exists and is usable. This is a slower but more reliable version of `has_header`. Most projects should use `has_header` instead of this one, which is only needed if you need to verify that the header can be included and compiled without errors.

Usage examples

```
dep = dependency('frobnitz')
cc.check_header('frobnitz.h', dependencies: dep)
```

18.3.4 `compiles(snippet, ...)` → boolean

Returns either `true` or `false` depending on whether the specified code snippet compiles successfully.

Positional arguments

snippet is the source code to be tested. This argument can either be a string holding the code or a file object.

Usage examples

```
# This will return false because the definition is missing.
cc.compiles('struct stat tmp;')

# This will return true.
cc.compiles('''#include<sys/stat>
struct stat tmp;''')
```

18.3.5 `compute_int(expression, ...)` → integer

Returns the value of the given expression as an integer.

Positional arguments

expression a single expression to evaluate.

Keyword arguments

guess is a best guess for the result. This is only used during cross compilation. Since cross compiled executables can not necessarily be run, Meson evaluates the results with an iterative search algorithm. Since cross compiled binaries can generally not be executed, the search is implemented by guessing a value, constructing a program that fails to compile if the value is not the result of the given computation and succeeds if it is. The system will try different values until it finds the correct one.

high specifies the upper bound for the iterative search result. If the answer is larger than this value, this function will return incorrect results. The default value is 1024. The larger the search space is, the slower the test is, so the boundary should be kept as close to the assumed value as possible.

low specifies the lower bound for the value. The default value is -1024.

Usage examples

```
# Result is 7
cc.compute_int('1 + 2*3')
```

18.3.6 `find_library(name, ...)` → dependency object

Finds the given library on the system. Returns a dependency object that can be used in build targets. The result must be used in **dependencies** keyword

argument, not `link_with` as the latter is only for libraries that have been built as part of the project.

201

Positional arguments

name the name of the library. The name should be specified without the `lib` prefix. It will be automatically added when required by the platform. That is, the value should be `something` rather than `libsomething`.

Keyword arguments

dirs is an array of strings with additional directories where the library should be looked for.

disabler is a boolean defaulting to `false`. If set to `true` causes a disabler object to be returned for failed lookups instead of a dependency object.

has_headers is a list of header file names. In order for the library lookup to be successful, all the given headers must all be found. Internally calls `has_header` on the inputs one by one.

required is a boolean defaulting to `true` causing failed library lookups to exit with a hard error. If set to `false` a non-found dependency object will be returned instead. Like all `required` keywords, this one also accepts a feature object.

In addition all keyword argument names that begin with `header_` will have this prefix removed and the result passed as a keyword argument to the corresponding `has_header` call.

Usage examples

```
# The portable way to find the C standard math library.
libm = cc.find_library('m', required: false)

# Look up a library and its header
l_dep = cc.find_library('frobicator',
                        has_headers: 'frobicate.h')
```

18.3.7 `get_id()` → string

Returns the name of the compiler as a string. Typical values include `gcc`, `clang` and `msvc`.

18.3.8 `get_linker_id()` \rightarrow `string`

Returns the name of the linker that the compiler will use.

18.3.9 `first_supported_argument(arguments)`
 \rightarrow `string` or `empty array`

Given the array of command line arguments, returns the first one for which the `has_argument` returns `true`. If none of the arguments are supported returns an empty array.

18.3.10 `first_supported_link_argument(arguments)`
 \rightarrow `string` or `empty array`

Given the array of command line arguments, returns the first one to which the `has_link_argument` returns `true`. If none of the arguments is supported returns an empty array.

18.3.11 `has_argument(argument)` \rightarrow `boolean`

Returns `true` if the compiler accepts the given command line argument and `false` otherwise. The argument is considered accepted if the compiler can successfully compile a test program while using the given argument. This test may be unreliable on some compilers, as they accept unknown command line arguments without an error. Meson tries its best to detect when a flag is not supported, but unfortunately this can not be made 100% reliable.

Usage examples

```
# Would return true on a gcc-style compiler.
cc.has_argument('-O2')

# Would return true on a msvc-style compiler.
cc.has_argument('/W3')
```

18.3.12 `get_supported_function_attributes(attributes)`
 \rightarrow `string array`

Returns an array of strings containing all attributes given as the argument that the compiler supports. Calls `has_function_attribute` internally.

18.3.13 `has_function(name, ...)` → `boolean`

Returns `true` if the given function exists.

203

Usage examples

```
# Result is true.
cc.has_function('fopen', prefix: '#include<stdio.h>')
```

18.3.14 `has_function_attribute(name)` → `string`

Returns `true` if the compiler supports a GNU style function attribute with the given name.

18.3.15 `has_header(name, ...)` → `boolean`

Returns `true` if the specified header exists. This is only a preprocessor existence check. If you want to test that the header is usable, use the `check_header` method instead.

Usage examples

```
# Almost always true.
cc.has_header('stdio.h')
```

18.3.16 `has_header_symbol(headername, symbolname...)` → `boolean`

Returns `true` if the specified header defines the symbol name given. More specifically it tests if the symbol can be found after including the header. The symbol may be defined in some other way (such as being a builtin), therefore this test may return `true` even if strictly speaking the symbol does not come from the header.

Positional arguments

headername is the name of the header to check.

symbolname is a string holding the symbol to check. What constitutes a “symbol” is language specific but usually contains things such as functions, variables, preprocessor symbols and types.

Keyword arguments

204

required is a boolean defaulting to **false**. If set to **true** and the symbol is not found, it will cause a hard error.

Usage examples

```
# A common function for getting file properties.
cc.has_header_symbol('unistd.h', 'fstat')

# This will cause a hard error.
cc.has_header_symbol('stdio.h', 'nonexisting', required: true)
```

18.3.17 `has_link_argument(argument)` → boolean

Same as `has_argument` but checks for linker arguments rather than compiler arguments. The way the linker is invoked depends on the toolchain. Unix-style toolchains invoke the linker via the compiler, and require all linker arguments to be prefixed with `-Wl,.` For Visual Studio Meson will automatically prepend the given argument with a `/LINK` argument.

Usage examples

```
# Would return true on a gcc-style compiler.
cc.has_link_argument('-Wl,-O1')

# Would return true on a msvc-style compiler.
cc.has_argument('/DEBUG')
```

18.3.18 `has_member(typename, membername, ...)` → boolean

Returns **true** if the given type (usually a struct) has a member with the given name.

Usage examples

```
# Stat is a struct holding information about
# file system entities and st_uid refers to the
# user id of the file's owner.
cc.has_member('struct stat', 'st_uid',
```

```

        prefix: '#include<sys/stat.h>')

# This returns false.
cc.has_member('struct stat', 'nonexisting',
              prefix: '#include<sys/stat.h>')

```

18.3.19 `has_members(typename, membernames, ...)` → boolean

Same as `has_member` but takes multiple member names. Returns `true` only if the type has all the listed members.

18.3.20 `has_multi_arguments(args, ...)` → boolean

Same as `has_argument` but takes many command line arguments and uses all of them in the test.

18.3.21 `has_multi_link_arguments(args, ...)` → boolean

Same as `has_link_argument` but takes many command line arguments and uses all of them in the test.

18.3.22 `has_type(typename, ...)` → boolean

Returns `true` if the argument defines a type.

18.3.23 `links(code, ...)` → boolean

Returns `true` if the given code compiles and links successfully.

Positional arguments

code is the program to be tested. Can either be a string holding the code or a file object.

Usage examples

```

# Returns false on systems that require specific
# linker arguments to use math functions.
snippet = '''#include<math.h>
int main(int argc, char **argv) {
    return sin(argc-1);
}'''

```

```
cc.links(snippet)
```

206

18.3.24 `run(code, ...)` → run result object

Tries to compile and run the given program. The return value can be used to check the outcome, including whether the compilation failed so the test could not be run.

Positional arguments

code is the code to be tested. Can be either a string holding the code or a file object.

Usage examples

```
# "code" is a simple application that always returns 0.

rc = cc.run(code)
rc.compiled()      # Returns true.
rc.returncode()    # Returns 0.
rc.stdout()        # Returns empty string.
```

18.3.25 `sizeof(typename, ...)` → integer

Returns the size of the specified type in multiples of size of `char`. This equals to bytes on all but the most esoteric of compiler toolchains.

If the type is unknown, the return value is -1.

Positional arguments

typename is the name of the type, such as `int`, `double` or `struct stat`.

Usage examples

```
# Almost always 4.
cc sizeof('int')
```

18.3.26 `symbols_have_underscore_prefix()` → boolean

Returns `true` if the system's C function name mangling scheme adds an underscore character before the symbol name. If you ever find yourself needing to

use this function, pat yourself on the back, for you are a member of a highly exclusive group of people.

207

18.3.27 `version()` → `string`

Returns the compiler's version number as a string.

18.3.28 Compilation check common arguments

Keyword arguments

args are command line arguments that are passed to compilation checks.

dependencies dependency objects to use. These must be external dependencies, using an internal dependency is a hard error.

include_directories list additional include directories to add to the header search path.

name is a freeform string that describes the current test and is printed when the check is run. Supported by `compiles`, `links` and `run` methods.

no_builtin_args when `true`, tells Meson not to add any builtin configuration options. Defaults to `false`.

prefix is a string that will be added to the beginning of test compilations. Typically it is used to specify `#includes` or type definitions.

18.4 configuration data object

A configuration data object is similar to a dictionary in that it stores a list of key and value pairs. The keys can only be strings and values can only be strings or integers. The main differences between dictionaries and configuration data objects are that configuration data object contains helper methods for common functionality and that configuration data objects are *mutable*.

Once a configuration data object is used as an argument to `configure_file`, it becomes immutable. Attempting to change its state will lead to program termination. This guards against the common problem where some values are accidentally set after file generation, not before it and thus would be lost.

18.4.1 `get(varname, default_value)` → `string`

Get back the value set for the given key.

Positional arguments

varname is the name of the value to look up.

default_value An optional value specifying the value to return if the object does not contain the specified key. If this is not set, then querying a missing value from the object is a hard error.

Usage examples

```
c = configuration_data()
c.set('key', 'value')
c.get('key')           # Result is 'value'
c.get('nonexisting')   # Hard error
c.get('nonexisting', 'fallback') # Result is 'fallback'.
```

18.4.2 `get_unquoted(varname, default_value) → string`

Get back the unquoted value of the given key.

Positional arguments

varname A string with the key value to look up.

default_value An optional value specifying the value to return if the object does not contain the specified key. If this is not set, then querying a missing value from the object is a hard error.

Usage examples

```
c = configuration_data()
c.set_quoted('key', 'value')
c.get('key')           # Result is '"value"'
c.get_unquoted('key')  # Result is 'value'.
```

18.4.3 `has(varname) → boolean`

Returns `true` if the given key has been set, `false` otherwise.

Positional arguments

varname A string with the key value to look up.

Usage examples


```
c = configuration_data()
c.set('key', 'value')
c.has('key')           # Result is true
c.has('nonexisting')   # Result is false
```

18.4.4 `merge_from(other)` → none

Takes all entries from the given argument and copies them to the current object. If a key is defined in both, the value in **other** is used and the current one is discarded.

Positional arguments

other is the configuration data object to copy entries from.

Usage examples

```
c1 = configuration_data()
c2 = configuration_data()
c1.set('key1', 'value1')
c1.set('key2', 'old_value')
c2.set('key2', 'new_value')
c2.set('key3', 'value3')
c1.merge_from(c2)
c1.get('key1')           # Result is 'value1'
c1.get('key2')           # Result is 'new_value'
c1.get('key3')           # Result is 'value3'
```

18.4.5 `set(varname, value)` → none

Sets the entry **varname** to the given value.

Positional arguments

varname A string containing the key to set.

value The value to set.

Usage examples

```
c = configuration_data()
c.set('key', 'value')      # Entry 'key' is now set to 'value'
c.set('key', 'value2')     # Entry 'key' is now set to 'value2'
```

18.4.6 set10(varname, value) → none

Sets the entry **varname** either to 1 or 0 depending on whether the given value is **true** or **false**. This method is convenient for defining e.g. C preprocessor definitions, which use numbers instead of boolean types.

Positional arguments

varname A string containing the key to set.

value The value to set, must be a boolean.

Usage examples

```
c = configuration_data()
c.set10('key1', true)      # Entry "key1" is now set to 1
c.set10('key2', false)     # Entry "key2" is now set to 0
```

18.4.7 set_quoted(varname, value) → none

Sets the entry **varname** to the given string but also quotes it with double quotes according to the C language quoting rules.

Positional arguments

varname A string containing the key to set.

value The value to set, must be a string.

Usage examples

```
c = configuration_data()
c.set_quoted('key', 'value') # Entry "key" is set to "value"
c.set('key', 'false')       # Identical to the previous line
```

18.5 custom_target object

211

A custom target object is the return value of the `custom_target` function. It is typically used to build further targets, especially when generating source code. Another use case of custom targets is to generate runnable scripts. Thus a custom target can be used as if it were an executable target. It is the responsibility of the user to ensure that the output is runnable when using a custom target as an executable. Meson can not detect this reliably, so it just assumes that the result can be executed directly.

Custom target objects can be indexed with the `[]` operator to access the individual output files.

18.5.1 full_path() → string

Returns a string with an absolute path pointing to the output file. Note that if you want to use the object as an argument to a different command, you should use the object directly in the argument array. Meson will automatically expand the path and set up a dependency between the two targets. This can also be called on individual outputs accessed via indexing, like this: `some_target[0].full_path()`.

18.5.2 to_list() → array

Returns an array of opaque objects referring to the individual output files of this target. Typically they are used as inputs to other targets and commands.

18.6 dependency object

A dependency object encapsulates a build dependency, typically a library with its corresponding headers. It can be either an external dependency as found by the `dependency` function or an internal one created with `declare_dependency`.

A dependency consists of four things: compiler flags, linker flags, libraries to link against, include directories and sources. Source files are compiled as part of the target as if they were listed in the target's sources. If any of the sources are headers generated during build, Meson will ensure that they are all generated before any source code using this target is compiled.

18.6.1 as_system() → dependency object

Returns a new dependency object that is otherwise identical to the current object, except that it is marked as a system dependency.

18.6.2 `found()` → `boolean`

212

Returns `true` if the dependency was found. Internal dependencies are always considered found.

18.6.3 `get_variable(...)` → `string`

Gets the value of the given variable from the dependency. These may contain additional information needed to configure builds, such as directory names for installing plugins and data files. Typically you only need to define `default_value`, but sometimes the variables have different names with different backends. In these cases you can override the name to use with the keyword arguments.

Keyword arguments

`cmake` is the variable name to use with a CMake dependency.

`default_value` is the default variable name to use if a tool-specific one has not been specified.

`configtool` is the variable name to use for dependencies that provide their own configuration tools rather than using something standard like Pkg-config. Some tools do not provide for this, in which case trying to query the value is a hard error.

`pkgconfig` is the variable name to use with a Pkg-config dependency.

`pkgconfig_define` can be used to override pkg-config variables during lookup. It behaves identically to the `define_variable` keyword argument in method `get_pkgconfig_variable`.

`internal` is the variable name to use for internal dependencies, that is, those created with `declare_dependency`.

18.6.4 `is_system()` → `boolean`

Returns `true` if the dependency was looked up as a system dependency. See Section 19.16 for details.

18.6.5 `name()` → `string`

Returns the name of the dependency or `internal` for internal dependencies.

18.6.6 `get_pkgconfig_variable(name, ...)` → `string`

Obtains the given variable from the dependency's pkg-config definition. Errors out if the dependency has not been found via pkg-config. Note that this method is deprecated and you'd probably want to use `get_variable` instead.

Positional arguments

name is the name of the variable to look up.

213

Keyword arguments

default is a value that will be returned if the variable is not defined.

define__variable can be used to override variables defined in the pkg-config file. This argument is an string array with consecutive key-value pairs.

Usage examples

```
dep.get_pkgconfig_variable('prefix')      # Returns '/usr'
dep.get_pkgconfig_variable('prefix',
    define_variable: ['prefix', '/tmp']) # Returns '/tmp'
```

18.6.7 `type_name()` → string

Returns the name of the mechanism (such as `pkgconfig`) used to look up this dependency or `internal` for internal dependencies.

18.6.8 `version()` → string

Returns the version of the dependency as a string or `unknown` if it can not be determined.

18.6.9 `partial_dependency(...)` → dependency object

Creates a new dependency object that consists of a subsection of this dependency's information. Used for special cases, such as when building extension modules for programming languages where you need the headers but you must *not* link against the library.

Keyword arguments

compile_args is a boolean specifying whether to add compile args to the result. Defaults to `false`.

includes is a boolean specifying whether to add include directories to the result. Defaults to `false`.

link_args is a boolean specifying whether to add link args to the result. Defaults to `false`.

Table 18.1: Results of setting the environment variable **E** with an environment object on different operating systems.

Method call	Old value	Unix	Windows
<code>.set('E', '1')</code>	any	1	1
<code>.set('E', '1', '2')</code>	any	1:2	1;2
<code>.set('E', '1', '2' separator: '_')</code>	any	1_2	1_2
<code>.append('E', '1')</code>	0	0:1	0;1
<code>.append('E', '1')</code>	none	1	1
<code>.prepend('E', '1')</code>	0	1:0	1;0
<code>.append('E', '1', '2')</code>	0	0:1:2	0;1;2
<code>.append('E', '1', separator: '_')</code>	0	0_1	0_1

links is a boolean specifying whether to add linked libraries to the result. Defaults to **false**.

sources is a boolean specifying whether to add sources to the result. Defaults to **false**.

18.7 environment object

Environment object is used to store information on how environment variables should be set when executing external commands and tests. Note that this object is *mutable*, you can change its state unlike almost every other Meson object. The way variables set with these commands interact with existing variables is outlined in Table 18.1.

18.7.1 `append(varname, values, ...)` → **none**

Positional arguments

varname is the name of the environment variable to change.

values are one or more strings specifying the values to append.

Keyword arguments

separator is the character used to separate the individual values. If not specified defaults to `;` on Windows and `:` on all other systems.

18.7.2 `prepend(varname, values, ...)` → `none`

This method is identical to `append` with the exception that it adds the values at the beginning of the given variable rather than at the end.

215

18.7.3 `set(varname, values, ...)` → `none`

This method is identical to `append` except that it discards any old value that the variable may have had.

18.8 `external_library` object

An external library is identical to an external dependency object except that its return value for `type_name` method call is `library`.

18.9 `external_program` object

An external program is the return value of `find_program` and represents a command that can be run. Note that the command may contain multiple parts and be different on different platforms. A script that is run with an interpreter might be `['python', 'myscript.py']` on Windows and `['myscript.py']` on Unix platforms.

18.9.1 `found()` → `boolean`

Returns `true` if the program has been found. Attempting to execute a not found executable object is a hard error.

18.9.2 `path()` → `string`

Returns an absolute path to the first part of the command. That is, if the command is a string, returns a string pointing to the interpreter binary. Don't use this when passing the object to a different command, pass the object itself directly instead.

18.10 `generator` object

A generator object is the return value of the `generator` function. It defines a rule of running the same operation on a bunch of input files to produce one or more output files. Generator outputs do not live on their own, they must be

passed to a build target to be ultimately created. This is because the rule writes the output files to the target's private directory.

216

18.10.1 `process(files, ...)` → output files

Tells Meson to process the given files with the current generator's rule.

Positional arguments

files are the input files to be processed, each one separately.

Keyword arguments

extra_args is an array of strings. If the generator has an `@EXTRA_ARGS@` template string in its argument list, that will be replaced with the specified arguments.

preserve_path_from tells Meson to preserve some path segments from the input filename. By default the file is written to the output directory, that is, if the given file name is `subdir/file.in`, the output will be `private_dir/file.out`. If the value is `meson.current_source_dir()` instead, the output file name will change to `private_dir/subdir/file.out`.

Usage examples

```
g = generator(...)

# Convert files.
out_files = g.process('file1.in', 'file2.in')

# Use the result.
executable('prog', 'prog.c', out_files)
```

18.11 `host_machine` object

This object is identical to the `build_machine` object, except that it represents the host machine. That is, the machine where the compiled program will be run on. The host machine only differs from the build machine when cross compiling. When compiling natively, the build and host machine objects are identical.

18.12 meson object

217

The **meson** object provides functionality to query state of the system and current build setup. It can also be used to define scripts to be run during different phases of the build.

18.12.1 `add_dist_script(executable, args) → none`

Adds a command to be run during when creating the **dist** target, that is, a release tarball. This method can be called multiple times and the scripts will be run in the order specified. This method can not be called from a subproject, only the master project. Calling it from a subproject is a hard error.

Positional arguments

executable is the script or command to execute. It may be a script, a **file** object or the output of **find_program**. Note that Meson will invoke the script from the *staging directory* after the first phase of dist has successfully finished.

args one or more strings that will be passed as arguments to the command.

Usage examples

```
ds = find_program('dist_postprocessor.py')
meson.add_dist_script(ds)
```

18.12.2 `add_install_script(script, args) → none`

This method is identical to **add_dist_script** except that it is run after install rather than dist. When the script is run, Meson sets environment variables as specified in Table 9.2 on page 106.

18.12.3 `add_postconf_script(script, args) → none`

This method is identical to **add_dist_script** except that it is run after the configuration phase of Meson has successfully finished.

18.12.4 `backend() → string`

Returns a string specifying the backend currently in use. Can be one of **ninja**, **vs2010**, **vs2015**, **vs2017** or **xcode**.

18.12.5 `build_root()` → string

218

Returns a string with an absolute path to the root of the current build tree. When called from a subproject will still return the top of the build tree. This method is rarely needed, most use cases are better served by `current_build_dir` instead.

Usage examples

```
# Returns a string like /home/username/myproject/build
bd = meson.build_root()
```

18.12.6 `can_run_host_binaries()` → boolean

Returns `true` if the binaries generated for the host can be run directly. This returns `true` when not doing cross compilation, when an exe wrapper has been defined and when the executables are directly runnable for other reasons. As an example the 64 bit version of Windows can run 32 bit Windows binaries natively.

18.12.7 `current_build_dir()` → string

Returns a string with an absolute path to the current build directory.

Usage examples

```
# Returns a string like /home/username/myproject/build/src
sd = meson.current_build_dir()
```

18.12.8 `current_source_dir()` → string

Returns a string with an absolute path to the current source directory.

Usage examples

```
# Returns a string like /home/username/myproject/src
sd = meson.current_source_dir()
```

18.12.9 `get_compiler(language, ...)` → compiler object

Returns a compiler object for the given language. Typically this is done to run configuration tests such as checking for existing headers.

Positional arguments

language the language to request.

219

Keyword arguments

native is a boolean specifying whether Meson should return the native or the cross compiler. If not defined the system will return the cross compiler when cross compiling and the native compiler otherwise.

Usage examples

```
# Get the C compiler
cc = meson.get_compiler('c')

# Get the native compiler even when cross compiling.
cc = meson.get_compiler('c', native: true)
```

18.12.10 `get_cross_property(name, fallback) → object`

Gets the value of a property as defined in the cross file. Return value can be anything defined in the cross file, typically a string, integer or a string array. Note that in most cases you'd want to use `get_external_property` instead as it works transparently with both cross and native files.

Positional arguments

name is a string holding the property name.

fallback an optional value to return if the specified property is not defined in the cross file or when not cross compiling. Requesting a missing cross property without a fallback is a hard error.

Usage examples

```
# Cross file has this:
#
# [properties]
# key = 'value'

# Returns 'value'
v = meson.get_cross_property('key')
```

```
# Hard error
v = meson.get_cross_property('nonexisting')

# Returns 'fallback'
v = meson.get_cross_property('nonexisting', 'fallback')
```

18.12.11 `get_external_property(name, fallback, ...)` → object

Behaves like `get_cross_property` but works transparently with both native and cross files.

Positional arguments

name is a string holding the property name.

fallback an optional value to return if the specified property is not defined.

Requesting a missing property without a fallback is a hard error.

Keyword arguments

native is a boolean specifying whether to get the information from the native file or the cross file. If not specified Meson will look up the property from the cross file when cross compiling and the native file when not cross compiling.

18.12.12 `has_exe_wrapper()` → boolean

This method is deprecated, use `can_run_host_binaries` instead.

18.12.13 `install_dependency_manifest(file_name)` → none

Installs a *dependency manifest* to the given file name. This manifest contains all subprojects, their versions and licenses in a machine readable form.

18.12.14 `is_cross_build()` → boolean

Returns `true` when cross compiling and `false` otherwise.

18.12.15 `is_unity()` → boolean

Returns `true` if doing a unity build and `false` otherwise. This is different from merely getting the value of the `unity` option. That option tells the global

value of the option, this method tells whether unity build is being used in *this* subproject.

221

18.12.16 `override_dependency_program(name, dependency) → none`

Overrides the result of subsequent dependency lookups to return the given dependency object. After this call all lookups with the given name either return that object or fail due to e.g. unsatisfiable version requirements. Trying to override a dependency that has already been successfully found is a hard error.

Positional arguments

name is a string specifying the dependency name, such as `glib-2.0`.

dependency is the dependency object to return, which can be the return value of `dependency` or `declare_dependency`.

18.12.17 `override_find_program(name, executable) → none`

Overrides the result of subsequent `find_program` calls so that looking up a program with the given name returns the executable given as the second argument. Trying to override a program name that has already been successfully looked during this Meson run is a hard error. This is done to ensure that only one executable is used in all projects.

This method is typically used to override code generators and the like in subprojects, so that all projects using them get the same specified programs.

Positional arguments

name is a string specifying the name to override.

executable is the program object that should be returned. Can be the result of `find_program`, `configure_file` or `executable`. If using an executable then there is an extra limitation that the overridden program may not be executed during configure time.

Usage examples

```
conv = find_program('custom_converter')
meson.override_find_program('converter', conv)

# Later in a different file
```

```
# conv will be custom_converter, not plain converter
conv = find_program('converter')
```

222

18.12.18 `project_licenses()` → string array

Returns a string array holding the current project's license(s) as set in the `project` function.

18.12.19 `project_name()` → string

Returns a string holding the current project's name as set in the `project` function.

18.12.20 `project_version()` → string

Returns a string holding the current project's version as set in the `project` function.

18.12.21 `source_root()` → string

Returns a string with an absolute path to the root of the current source tree. When called from a subproject will still return the top of the master project. This method is rarely needed, most uses are better served by `current_source_dir` instead.

18.12.22 `version()` → string

Returns a string holding the version of Meson currently in use.

Usage examples

```
# Returns a string like '0.55.0'
sd = meson.version()
```

18.13 `python_installation` object

This object is created by the `python` module as documented in Section 20.8. The installation object behaves like an external program. It can be invoked with `run_command` or used to generate custom targets. It has additional methods for creating extensions.

18.13.1 `extension_module(name, sources, ...)`
→ module object

This method builds a Python extension module according to the naming conventions of the current platform. This method is identical to the `shared_module` function except that you can't define the file name's prefix or suffix.

The method accepts one additional keyword argument, `subdir`, which specifies the installation directory relative to the system's extension module directory. Defining both this keyword argument and `install_dir` is a hard error.

18.13.2 `dependency()` → dependency object

Returns a dependency object needed to build extension modules against the given installation.

18.13.3 `install_sources(files, ...)` → none

Installs Python source files. Positional arguments are the same as they are for `install_data`.

Keyword arguments

pure specifies whether compiled modules should be installed in the same directory as their Python source code. The default is `true`, keeping the two file types in separate directories.

subdir has the same meaning as in the `extension_module` method.

In addition this method accepts all the same keyword arguments as the function `install_data`.

18.13.4 `get_install_dir(...)` → string

Returns the directory where Python source files get installed. This method accepts the same keyword arguments as `install_sources`.

18.13.5 `language_version()` → string

Returns the current installation's Python version in the format `major.minor`.

18.13.6 `get_path(name, fallback)` → string

Gets the given path. This is equivalent to calling `sysconfig.get_path` in Python.

Positional arguments

224

name is a path name to look up. The list of existing path names depend on the Python installation, but can include values such as **stdlib** and **data**. **fallback** is an optional fallback value that is returned if the path name was not found. If a name is not found and fallback is not set, Meson will exit with a hard error.

18.13.7 `has_path(pathname)` → `boolean`

Returns **true** if the Python installation has the given path.

18.13.8 `get_variable()` → `string`

This method is identical to `get_path` except that it looks up variables rather than paths in the `sysconfig` module.

18.13.9 `has_variable()` → `boolean`

This method is identical to `has_path` except that it looks up variables rather than paths in the `sysconfig` module.

18.14 `run_result` object

Run result stores information on an external process executed during configure time. These include `run_command` and checks invoked via the compiler object.

18.14.1 `compiled()` → `boolean`

This method is only available on objects created by compiler checks. Returns **true** if the code built successfully. If **false** the remaining methods return unspecified data.

18.14.2 `returncode()` → `integer`

Gets the return code of the program. The established standard is that the value 0 means success and all other values mean failure.

18.14.3 `stderr()` → `string`

Returns all text the program wrote to standard error.

18.14.4 `stdout()` → `string`

Returns all text the program wrote to standard out.

225

18.15 `source_configuration` object

This object contains the outcome of evaluating a source set. It holds sources and dependencies that can be used to build targets.

18.15.1 `sources()` → `source list`

Returns all sources in the current object.

18.15.2 `dependencies()` → `dependency list`

Returns all dependencies in the current object.

18.16 `source_set` object

A `source_set` object is used to convert long and convoluted `if/else` branches into simpler form. A typical use case is implementing a function with an existing library or, if it is not available, with fallback functionality (which is usually less performant). These objects are created with the `source_set` module as described in Section 20.12.

This object has two distinct phases. First you define the different sources and conditions for their usage. Then you pass in a configuration object to obtain the final set of sources and dependencies to use in a target. This object is mutable in the first phase, but becomes immutable once you pass in a configuration object. Trying to add new sources after that is a hard error.

18.16.1 `add(sources_and_deps, ...)` → `none`

This method adds new sources and dependencies to the set, either conditionally or unconditionally. This method can thus be called only with positional arguments or only with keyword arguments. Having both is a hard error.

Positional arguments

`sources_and_deps` consists of source and dependency objects. These are added unconditionally to this set's list of sources and dependencies.

Keyword arguments

226

when lists this set's conditions that are either strings or dependency objects. The condition is considered to be **true** if all dependencies represent dependencies that have been found and if all strings have been set to **true** in the condition object used as the argument in the **apply** method.

if_false lists the sources and dependencies to add if the condition is not **true**.

if_true lists the sources and dependencies to add if the condition is **true**.

18.16.2 `add_all(source_sets, ...)` → none

Joins the given source sets to the current source set. Positional and keyword arguments work the same way as in the **add** method.

Keyword arguments

when is a condition similar to the one in **add**.

if_true lists the source sets to add if the condition is **true**. Note that the conditions defined in the arguments are still in affect, that is, their individual entries are only used if both this condition and the argument's condition are **true**.

18.16.3 `all_sources()` → source list

Returns a list of all sources added to this set and which do not have a non-found dependency object in their condition statement.

18.16.4 `all_dependencies()` → dependency list

Returns a list of all dependencies added to this set and which do not have a non-found dependency object in their condition statement.

18.16.5 `apply(configuration, ...)` → source configuration

Evaluates all sources and dependencies added to this source set with the given configuration object and returns all that matched as an object. The return value is typically used to define build targets.

Positional arguments

configuration is either a dictionary or a **configuration** object. Each conditional string defined with an **add** method is checked against this configura-

tion object. The sources and dependencies that match the corresponding value in the dictionary are placed in the return value.

227

Keyword arguments

strict defines whether missing values should be treated as **false** or a hard error. The default value is **true**, which means that any rule that uses a variable not defined in the configuration object is a hard error.

Usage examples

Suppose we have the following conditional build setup that has both an optional dependency as well as a toggle option.

```
d1 = dependency(...)
all_sources = ['always.c']
all_deps = []
conf = configuration(...)

if d1.found()
    all_deps += d1
    all_sources += files('dep_impl.c')
else
    all_sources += files('fallback_impl.c')
endif

conf.set('use_widget', get_option('widget'))
if get_option('widget')
    all_sources += files('use_widget.c')
endif

executable(..., all_sources,
              dependencies: all_deps)
```

The same functionality can be achieved with less effort by using a source set.

```
d1 = dependency(...)
conf = configuration(...)
smod = import('sourceset')
sset = smod.source_set()
sset.add('always.c')
sset.add(when: d1,
```

228

```
        if_true: 'dep_impl.c',
        if_false: 'dep_fallback.c')
sset.add(when: 'use_widget',
        if_true: 'dep_widget.c')
sconfig = ss.apply(con)
executable(..., sconfig.sources();
           dependencies: sconfig.dependencies())
```

18.17 target_machine object

This object is identical to the `build_machine` object, but it refers to the target machine. Most projects don't need to care about target machines, only about build and host machines. The target machine is only relevant for cross compiling cross compilers. The different machines are documented in Chapter 12.

18.18 subproject object

This object encapsulates the result of a subproject invocation and can be used to extract information from the subproject

18.18.1 found() → boolean

Returns `true` if the subproject executed successfully.

18.18.2 get_variable(name) → object

Extracts a variable from the subproject and brings it to the current project. The result can be used as if it was generated in the current project.

Usage examples

```
# In a subproject
subproject_variable = 3

# In the master project
sp = subproject(...)
master_variable = sp.get_variable('subproject_variable')
# master_variable has the value 3.
```

Chapter 19

Function reference

This chapter describes all functions that can be called from a build definition file. There is one additional function, `option`, that can only be called from an option file. It is documented in Section 10.3.

19.1 `add_global_arguments(args, ...)` \rightarrow none

This function is used to easily add compilations flags for *all* compiler invocations for a given language, even for subprojects. Most projects should not use this function, but `add_project_arguments` instead. Calling this function from a subproject is a hard error, because subprojects are not allowed to change the master project's state.

A typical use case for this function is enabling custom debug functionality globally from the master project. As the name implies, multiple calls to this function adds both sets of arguments. It is not possible to unset arguments from a build file.

Positional arguments

args one or more string arguments that will be passed to all compiler invocation for the given language.

Keyword arguments

language contains a list of one or more languages that the arguments should be added to. Using a language not enabled by `project()` or `add_languages` is a hard error.

native is a boolean controlling whether the flags should be used for native or cross compilation. If omitted this will default to native compilation when not cross compiling and cross compilation otherwise.

Usage examples

```
add_global_arguments('-DCLIKE', languages: ['c', 'cpp'])
# The argument is added to C and C++ compilations

add_global_arguments('-DIS_CROSS',
                     languages: 'c',
                     native: false)
# This argument is only used when cross compiling C.
```

19.2 `add_global_link_arguments(args, ...)` → none

This function behaves identically to `add_global_arguments`, except that the arguments are used during linking instead of during compilation. Similarly most projects should use `add_project_link_arguments` instead of this function.

Most projects don't need to deal with linker arguments at all. Meson provides builtin features for most things. Linker arguments are most commonly needed for during cross compilation, and those should be defined in the cross file rather than in build files.

Positional arguments

args A string or string list of arguments consist of zero or more strings that will be passed to the linker invocation for the given language.

Keyword arguments

language contains a list of one or more languages that the arguments should be added to. Using a language not enabled by `project()` or `add_languages` is a hard error.

native is a boolean controlling whether the flags should be used for native or cross compilation. If omitted this will default to native compilation when not cross compiling and cross compilation otherwise.

Usage examples

```
# The argument is added to C and C++ links
add_global_link_arguments('-Wl,--gc-sections',
                           languages: ['c', 'cpp'])
```

231

19.3 add_languages(langs, ...) → none/boolean

Adds the given languages to the current project's list of languages. By default the behaviour is the same as if the language had been given in the `project` function. This function is used to enable optional and platform-specific languages, such as only enabling Java if compiling for Android or Objective C when compiling on Apple platforms.

Positional arguments

langs should be one or more strings of language names to add.

Keyword arguments

required can be set to **false** to indicate that the languages are optional. In this case Meson will return **true** if all the specified languages were found and **false** if any one of them was not found. The default value is **true** meaning that if the compiler for the given language is not found, Meson will exit with an error.

This keyword argument can also be given a **feature** option. The behaviour is the same, it merely allows you to tie the language's required status into a build option.

native if set to **true** look up a native compiler, that is, one that builds for the "build machine". If **true**, a cross compiler is searched when cross compiling and the normal compiler otherwise. If omitted, Meson will look up both compilers.

Usage examples

```
add_languages('objc')
# At this point Objective C has been found and is usable.

add_languages('nonexisting')
# This is a hard error
```

```
result = add_languages('nonexisting', required: false)
# result is false
```

19.4 add_project_arguments(args, ...) → none

This method behaves identically to `add_global_project_args` except that the flags are only used for the current project.

Usage examples

```
add_project_arguments('-DCLIKE', languages: ['c', 'cpp'])
# The argument is added to C and C++ compilations

add_project_arguments('-DIS_CROSS',
                      languages: 'c',
                      native: false)
# This argument is only used when cross compiling C.
```

19.5 add_project_link_arguments(args, ...) → none

This function behaves identical to `add_project_arguments`, except that the arguments are used during linking instead of during compilation.

Usage examples

```
# The argument is added to C and C++ links
add_project_link_arguments('-Wl,--gc-sections',
                          languages: ['c', 'cpp'])
```

19.6 add_test_setup(name, ...) → none

Creates a new test setup with the given name. It is enabled with a command line argument to Meson's test runner.

```
$ meson test --setup=mysetup
```


Positional arguments

name is a string specifying the name of the setup. Trying to specify the same setup multiple times is a hard error.

Keyword arguments

env is an environment object specifying environment variables to set when running the tests.

exe_wrapper defines an executable that will be used to invoke the test program. This can either be a self built executable or the output of **find_program**. This argument can also be an array, in which case the first element is the program and the rest are arguments given to the wrapper program before the actual test command.

gdb is a boolean. If set to **true** it means that the test setup should be run under GDB, which is the same as passing **--gdb** to **meson test**.

timeout_multiplier is a number used to multiply the default timeout value. Typically wrapper programs make test execution slower. This argument can be used to override timeouts globally. This is identical to passing the **--timeout-multiplier** argument to **meson test**.

is_default sets the specified setup as the default.

Usage examples

```
valgrind = find_program('valgrind')
supp_file = meson.current_source_dir() / 'suppressions.txt'
supp_argument = '--suppressions=' + supp_file
env = environment()
env.set('IS_VALGRIND', '1')
add_test_setup('valgrind',
               exe_wrapper: [valgrind, supp_argument],
               timeout_multiplier: 50
               env: env)
```

Tests can be run under Valgrind with this command:

```
$ meson test --setup=valgrind
```

19.7 alias_target(name, deps) → none

Creates a new target that by itself does nothing but ensures that the specified targets are built. A common use case for this is that the project has some helper

tools that are not built by default but which are needed every now and then. Thus building this target (just as one would “build” a `run_target`) can be used to provide a convenient shorthand to building all of them in one go.

Positional arguments

name is the name to give this target.

deps are one or more target objects that are built when this target is built. They can be either build targets or run targets.

19.8 `assert(condition, message) → none`

If the given condition evaluates to `false`, prints the message and exits Meson with a hard error.

Positional arguments

condition is a boolean specifying whether to exit or not.

message is a freeform string to be printed in case of error. If not specified, Meson will print a prettyprinted version of the condition statement instead.

Usage examples

```
# Does nothing.
assert(1 == 1, 'Equality comparison is broken.')

# Exits the program.
assert(not(1 == 2), 'Your universe has weird math axioms.')
```

19.9 `benchmark(name, executable, ...) → none`

This function behaves identically to the function `test` except that it defines a benchmark to be run with `meson benchmark` command rather than a test. See the reference documentation of the `test` function in Section 19.48 for details. The only difference is that there is no `is_parallel` keyword argument, because benchmarks are only run strictly sequentially to minimise timing jitter.

19.10 both_libraries(name, sources, ...)

→ library object

235

Builds the given sources both as a shared and as a static library. If possible, the source code is compiled only once and the same object files are used for both targets. If this can't be done because, for example, the compilations would require different command line arguments, the sources are built separately for each target.

The return value of this function represents a shared library, but it also has methods `get_shared_lib` and `get_static_lib` to access the shared and static libs, respectively.

Both positional and keyword arguments for this function are the same as for `shared_library` and `static_library`.

Usage examples

```
libs = both_libraries(...)

# This is static linked.
executable(..., link_with: libs.get_static_library())

# This is shared linked.
executable(..., link_with: libs.get_shared_library())

# As is this
executable(..., link_with: libs)
```

19.11 build_target(name, sources, ...)

→ build target

Constructs a new build target of the specified type. With the exception of `target_type` all keyword arguments listed here can also be passed to other build target functions such as `executable`. They are documented here, whereas each individual function only documents its additional keyword arguments. The most common usage for this function is building a library either as shared or static with a toggle of its own rather than following the value of the `default_library` option.

Positional arguments

name the base name of this target, meaning the output file name without any prefixes or file extensions.

sources are source files to be used in this target. Sources can also be defined with the **sources** keyword argument. If both are defined, all files from both of them are used.

Keyword arguments

<language>_pch specifies the files to be used as precompiled headers for this target. This argument is per-language, so to define a precompiled header for C++, you'd use the keyword argument **cpp_pch**. Some toolchains do not permit more than one precompiled header per target, trying to specify more than one may lead to build failures. The value may consist of at most one header and one source file. If the source file is not specified, it will be autogenerated when necessary.

<language>_args is an array of strings that should be used as command line arguments when compiling code in the specified language.

build_by_default specifies whether this target is built when the default build command is invoked. The default value is **true**.

build_rpath is a list of strings representing directories that are added to the target's rpath when it is in the build directory. These values are removed from the file when it is installed.

dependencies list all dependencies needed to build this target.

extra_files contains a list of files that should be considered part of this target even though they are not used directly. This option does not affect building in any way, just causes these files to be listed in the given target in IDEs that hide the file system and instead group files by top level targets.

gnu_symbol_visibility is a string specifying how symbols are exported from the current target. This keyword argument is ignored on compilers that do not support GNU symbol visibility. The value can be either empty or one of **default**, **internal**, **hidden**, **protected** or **inlineshidden**. The last one is the same as **hidden** but also hides C++ implicit functions.

implicit_include_directories can be set to **false** to prevent Meson from adding the current source and build directories to the header search path automatically. The default value is **true**.

include_directories lists directories that should be added to the header search list. The values can either an **include_directories** object or strings. The latter is identical to passing the string to **include_directories** function and using the result.

install is a boolean specifying whether this target is installed. The default value

is `false`.

install_dir is a string specifying the install directory. The default value depends on the target type, for example executables are installed to the value specified in option `bindir`.

install_mode specifies the properties of the installed file. This argument consists of up to three entries. The first one is a string specifying the Unix permission bits of the file. The second and third ones specify the user and group ids of the file, respectively. They can be specified either as a string or as a number.

install_rpath is an array of strings specifying the rpath entries that this target should have after install. They are not used when the binary is in the build dir.

link_args is an array of strings. These are passed to the linker when linking this target.

link_depends lists files and targets that are required to link the current target. If any of these dependencies change, Meson will rerun the link phase. The most common case for this is linker scripts.

link_language specifies which programming language should be used for linking the target. This is only necessary for projects that mix multiple languages. Meson tries its best to choose the correct link language, but there are cases where it is not done reliably.

link_whole is similar to **link_with** except it tells the linker to link the whole archive instead of only the parts needed to satisfy missing symbols. This is only meaningful for static libraries. Some toolchains do not support the concept of whole linking. In these cases Meson simulates the behaviour by adding all the object files of the link target to the current target.

link_with lists build targets (outputs of `static_library` or `shared_library`) that should be linked in the current target. The transitive dependencies of shared libraries are *not* added to the link line but for static libraries they are.

name_prefix is a string specifying the prefix to be used for this target. The default value is chosen according to the target type and platform. For example library targets usually have the prefix `lib` except on Windows where the prefix is an empty string.

name_suffix is a string specifying the file extension for this target. If omitted Meson will use a default value depending on the system and target type. For example an executable's suffix is `.exe` on Windows and empty on all other platforms.

native is a boolean specifying whether this target should be cross-compiled or native compiled. Typically you'd set this to `true` for targets that you want to execute during the build, such as source code generators. Trying

to install native built objects during cross compilation is a hard error.

objects lists prebuilt objects that should be linked to this target. This is only for objects whose source code you don't have. Objects built by this project should be added to sources.

override_options is an array of strings of type 'option_name=value' that can be used to override option values for this target.

target_type is a string which can be one of `executable`, `shared_library`, `shared_module` or `static_library`.

Usage examples

```
# A tool that is needed rarely and thus not built by default.
build_target('debugtool', 'debugtool.c',
             target_type: 'executable',
             build_by_default: false)

# A system tool that can only be run by root.
build_target('systemtool', 'systemtool.c',
             target_type: 'executable',
             install: true,
             install_dir: '/sbin',
             install_mode: ['rwx-----', 'root', 'root'])

# Defining the target type with a project option.
build_target(...,
             target_type: get_option('librarytype'))

# Build this target with C++ version 11 regardless
# of what the default language version is.
build_target(...,
             override_options: 'cpp_std=c++11')
```

19.12 `configuration_data()` → configuration data

Returns an empty configuration data object.

19.13 `configure_file(...)` → none

Generates an output file, usually a header file, with the given name in the current build directory. There are three different ways of creating the file. The first one

takes a template file as input, does text substitution and writes the result to the output file. The second one takes no input file but instead serialises the configuration data directly to the file. The third and final way generates the output file by executing a standalone script during configuration.

Keyword arguments

capture is a boolean. If set to **true**, Meson will grab the external script's standard output and write it to the output file. This is used for misbehaving programs that won't write their output to a file, but instead always print it to standard out.

command is an array specifying the command to run instead of doing template substitution. The arguments follow the same filename expansion rules as source generation described in Section 11.3.

configuration is a configuration object. All substitution values will be taken from this object and nowhere else. This may also be a dictionary object. It is internally converted to a configuration object and then used.

copy is a boolean. When set to **true** tells Meson to just copy the input file to the output file without doing any substitutions.

depfile is the name of a dependency file, that the generator script writes that lists all additional files it depends on. If any of those files changes, Meson will run a full reconfiguration.

encoding specifies the text encoding of the source and output files. Defaults to **utf8**.

format specifies the format of the template file. The default value is **meson** for Meson's internal format. Can also be set to **cmake** to replicate CMake's behaviour or to **cmake@** to replicate CMake but only replace variables marked with @ signs, not expansions of type **\${variable}**.

input optional input file(s) to use.

install a boolean specifying if the generated file should be installed to the directory specified in **install_dir**. If omitted the file will be installed if **install_dir** is specified and not installed otherwise.

install_dir optionally specifies the directory the generated file will be installed to.

install_mode behaves identically to the same keyword in a build target as described on page 237.

output is the output filename. If the file name contains the strings **@BASENAME@** or **@PLAINNAME@** they will be expanded according to the file name rules in Section 11.3.

output_format defines the format of the expanded preprocessor definitions. The default value is **c** causing the values to be prefixed with **#**. The value of **asm** is also supported, which causes the definitions to be prefixed with

%.

240

Usage examples

```
cdata = configuration_data()
cdata.set('key', 'value')

# The common case
configure_file(input: 'project-config.h.in',
               output: 'project-config.h',
               configuration: cdata)

# Generate output file from input data.
configure_file(output: 'project-config.h',
               configuration: cdata)

# Install a global configuration file.
configure_file(output: 'conffile',
               input: 'conffile.in',
               install_dir: '/etc',
               install_mode: ['rw-r--r--', 'root', 'root'])

# Generate a file with a script.
g = find_program('gen_script')
configure_file(output: 'outfile',
               input: 'infile',
               command: [g, '@INPUT@', '@OUTPUT@'])
```

19.14 custom_target(name, ...) → custom target

This function allows you to create fully custom build steps. The most common use case is generating source code for the build, but generating documentation and converting asset files is common.

Positional arguments

name is the name of this target. It does not necessarily have anything to do with output file(s), but it's considered good practice to keep these two close together.

Keyword arguments

build_by_default specifies whether this target should be built when the default build command is executed. The default value is **false**, meaning that the target will not be built unless explicitly requested. This can be done by requesting this target to be built or indirectly by building a target that depends on this target. If this argument is missing the target will be built by default if **install** is set to **true**.

build_always_stale is a boolean. When set to **true** causes this target to always be considered “out of date”. It will be rebuilt even if all its dependencies are up to date.

capture is a boolean. If set to **true**, Meson will grab the commands standard output and write it to the output file. This is used for poorly behaving programs that can’t be told to write their output to a file, but instead always print it to standard out.

console is a boolean designed to help integrate long running tasks. If **true**, this target will be run in isolation, that is, no other build task is run at the same time. It also disables console buffering, so everything the command prints is immediately shown. This option can not be used together with **capture** and attempting to do so is an unrecoverable error.

command is the actual command to be executed. This is usually an array with the first argument being the command to run, which can be the return value of **executable**, **find_program** or **custom_target**. The remaining items are the command line arguments to be used, either strings or generated targets. Strings are expanded according to the rules described in Section 11.3. Meson targets are automatically marked as dependencies of this target and converted to command line strings.

depend_files lists extra files (either strings, file objects or configure file output) that this command depends on but which are not passed to the command as arguments. This is usually because the command does globbing internally.

depends lists targets that this command depends on even though they are not listed on the command line. Like with **depend_files** this is usually because the command does some form of globbing internally.

depfile is the name of the **dependency file** that the command generates. This file must be generated in the same Makefile-esque format that compilers generate by default.

input contains the list of input files.

install causes the target to be installed when set to **true**. Requires **install_dir** to be specified also.

install_dir lists the directory to install the output file(s) to.

install_mode behaves identical to the same keyword in a build target as de-

scribed on page 237.

output the output file name(s).

242

Usage examples

```
prog = find_program(...)

# convert file.idl -> file.c
gen_c = custom_target('c_generator',
    input: 'file.idl',
    output: '@BASENAME@.c',
    command: [prog, '@INPUT@', '-o', '@OUTPUT@'])

# Generate source and header.
gen_files = custom_target('c_generator',
    input: 'file.idl',
    output: ['@BASENAME@.c', '@BASENAME@.h']
    command: [prog, '@INPUT@', '--outdir', '@OUTDIR@'])

# Convert texture and install it:
custom_target(...
    input: 'texture.png',
    output: 'texture.dds',
    command: ...,
    install: true,
    install_dir: 'share/mygame/textures')
```

19.15 declare_dependency(...)

→ dependency object

Creates a new dependency object that can be used to build targets. The result can be used interchangeably with other dependency objects, such as the return value of `dependency()`.

Keyword arguments

compile_args lists the compilation args needed to use this dependency. This should only be used for special arguments that can not be expressed with the other keyword arguments.

dependencies are other dependencies needed to use this dependency. Typically needed for cases such as when this dependency exposes some part of their dependencies (such as data types) and thus you need the original headers to be able to compile code.

include_directories lists the include directories in the same format as in a **build_target**.

link_args lists the arguments that should be passed to the linker. Like with **compile_args**, this should only be used for special arguments.

link_with lists targets that should be linked with just like in **build_target**.

link_whole lists targets (static libraries) that should be wholly linked in the same format as **build_target**.

sources is an array of source files that should be added to the target. A common case is to list generated headers needed to use this dependency. This will make Meson ensure that all of these files have been created before any source code is built.

variables is a dictionary of arbitrary string that can be queried from the generated dependency object. They behave just like the corresponding variables in external dependencies, like pkg-config. Typically these are defined to have the same entries as external pkg-config files so that the two dependencies can be used interchangeably.

version is a string specifying the version of this dependency.

Usage examples

```
lib = library(...)
inc = include_directories('include')

# Create a simple dependency.
my_dep = declare_dependency(link_with: lib,
                             include_directories: inc)

# Then use it in a different directory
executable(..., dependencies: my_dep)

# The above is identical to adding the individual
# items directly as keyword arguments.
executable(...,
            link_with: lib,
            include_directories: inc)
```

19.16 `dependency(name, ...)` → `dependency` object

244

Finds a dependency either from the system or by building a subproject. By default the system lookup is done in the “system native” way falling back to other methods if not found. The default lookup mechanism is Pkg-config, as it is the most widely used system in practice.

Positional arguments

name is a string specifying the name of the dependency to search for.

Keyword arguments

components can be used to specify CMake `COMPONENTS` to use during library lookup. This keyword argument is only available when looking up CMake dependencies.

default_options is an array of strings defining default options that should be used in case the dependency is provided by a subproject. The option values specified here override the default options set in the subproject’s `project` call. Note that they can only contain project options, not Meson options because those have already been set up.

disabler will cause, when set to `true`, a disabler object to be returned for not found objects. If set to `false`, which is the default, a non-found dependency object is returned instead.

fallback tells Meson how to obtain this dependency if it is built as a subproject. It consists of an array of one or two strings, the subproject name and a variable name. These should point to a variable in the subproject that holds the desired dependency object. If using only one string, then the subproject must specify which variable holds the dependency either via a wrap file or with `override_dependency`.

is_system is a boolean specifying whether the include directories defined by the dependency should be treated as “system directories”. This reduces compiler warnings, as most compilers do not print warnings that originate from system headers. Default value is `false`.

language specifies which programming language the dependency is for. This is needed for some dependencies that are used differently from, say, C++ and Fortran.

method is a string specifying how the method should be looked up.

native is a boolean specifying whether the dependency will be used for native or cross compilation. If unset defaults to native building when not cross

compiling and not native when cross-compiling.

modules is a list of strings for defining which subparts of the dependency should be used. Some large dependencies such as Qt and Boost ship a bunch of small individual components instead of just one main dependency. The way a dependency is split into modules is not specified, each dependency is free to choose their own module setup.

not_found_message is a string that is printed if the dependency is not found.

required is a boolean specifying whether this dependency is mandatory. Defaults to **true**. Failing to look up a mandatory dependency is a hard error.

static is a boolean telling whether Meson should look for a static or shared library version of the dependency. Not all backends support this argument.

version specifies version requirements for the dependency. This can be either a string or an array of multiple strings in the format required by **version_compare**. The exact format is described in string object's method reference in Section 17.6.12. The version is considered suitable if it passes each listed requirement.

Usage examples

```
# Basic lookup
sdl2_dep = dependency('sdl2')

# Use sdl2-config instead of pkg-config
sdl2_dep = dependency('sdl2', method: 'config-tool')

# Get Qt 5's GUI widget libraries only.
qt5widget_dep = dependency('qt5', modules: ['Widgets'])

# Version requirements
foo_dep = dependency('foo', version: '>=1.0.0')

# Return a disabler
some_dep = dependency('nonexisting', disable: true)

# Always returns a non-found dependency.
unfound_dep = dependency('', required: false)
```

19.17 `disabler()` → disabler object

Creates a disabler object.

19.18 `environment(initial_values)` → `environment` object

Creates a new environment object.

Positional arguments

initial_value is an optional dictionary object. If specified, its entries are added to the resulting environment object.

19.19 `executable(args, ...)` → `none`

Creates an executable, that is, a binary that can be run.

This function supports all the same keyword arguments as `build_target`. See Section 19.11 for further reference.

Keyword arguments

export_dynamic is a boolean specifying whether the symbols in the executable will be exported to the global symbol table. This property is also known as `rdynamic` in some toolchains.

gui_app is a boolean specifying that the executable should be treated as a graphical application rather than a command line executable. This distinction only exists on few platforms, chiefly Windows. It is ignored on other toolchains.

implib if `true` generates an import library which can be used to link other targets against this executable as if it was a library. This argument is only used on Windows and ignored on all other platforms.

Usage examples

```
# The common case.
executable('hello', 'hello.c',
          install: true)

# Lua is typically used by bundling it inside
# the main executable. In order to use Lua plugins,
# with dlopen(), we must export the symbols from
# the executable.
lua_lib = static_library(...)
```

```
executable(...  
    link_with: lua_lib,  
    export_dynamic: true)
```

19.20 `error(message)` → none

Prints the given message string and exits the current Meson invocation with a hard error.

Positional arguments

message is a string with the message to be printed.

Usage examples

```
error('An <unexpected event> has happened.')
```

19.21 `find_program(names, ...)` → program object

Finds a program with the given name from the system. If the program name has been overridden with `meson.override_find_program`, the value set in that call is returned.

In addition to programs this code will also look up executable scripts in the path and in the current source directory. The lookup parses Unix shebang lines of scripts on platforms that do not support them natively. In practice this means Windows.

Positional arguments

names are one or more strings with command names to look up. They should not have an `.exe` suffix even on Windows. Meson will look up the commands one by one and returns the first one that could be found.

Keyword arguments

dirs lists one or more additional directories, such as `/usr/local/bin`, where the program should be looked up in.

disabler is a boolean. When set to `true` Meson will return a disabler object rather than a not found object when the program can't be found.

248

native tells whether the program should be looked up in the cross file (when set to **false**) or from the system (when **true**). Default value is **false**.
required a boolean specifying whether the failure to find a program is a hard error. The default is **true**.

Usage examples

```
# The default case. Exits with a hard error if the program
# can not be found.
prog = find_program('prog')

# The program is optional
notfound_prog = find_program('nonexistin', required: false)

# This program has different names on different platforms.
prog = find_program('prog', 'a_prog', 'the_prog')

# Look up a program from directories not usually in path.
prog = find_program('prog',
                    '/usr/local/bin/prog',
                    '/sbin/prog')
```

19.22 files(list of files) → array of file objects

Converts the given input file names into file objects. They are more convenient to use than string arrays. File objects are guaranteed to exist on the file system. That is, calling this function with a string pointing to a non-existing file is a hard error. In addition file objects remember their locations. You can use them in any directory and Meson will automatically expand the object to point to the original file.

Positional arguments

list of files are one or more strings representing file names to convert to file objects. Relative paths are interpreted from the current source directory.

Usage examples


```
# The common case.
f = files('foo.c', 'bar/bob.c')

# Absolute paths are also supported.
f = files('/usr/share/something/file.c')
```

19.23 generator(executable, ...)

→ generator object

Creates a new generator object, which encapsulates a command to do file transformation. It can be used to apply the same operation to a bunch of input files.

Positional arguments

executable is the program that will be called to do the transformations. Can be either the result of `find_program` or an `executable` or a `custom_target`.

Keyword arguments

arguments list all the command line arguments that will be used when calling the program. Special strings will be replaced with real values as described in Section 11.3.

capture is a boolean. When set to `true` it tells Meson to take the program's output and write that to the output file. This is used to deal with uncooperative programs that always their results to standard output rather than to an output file.

depfile is an optional file name describing the file where the called program will write dependency information. The output of this file must be in the same Makefile-esque format that compilers use. If the program does not create a depfile, this keyword argument should be omitted.

output is a template string describing how the output file name can be derived from each input file. The substitution uses the rules defined in Section 11.3.

Usage examples

```
# A simple source generator.
# Converts file.idl to file.c
g = generator(gen_exe,
```

250

```

arguments: ['@INPUT@', '-o', '@OUTPUT@']
output: '@BASENAME@.c')

# Create both source and header files
g = generator(gen_exe,
arguments: ['@INPUT@', '--outdir', '@BUILD_DIR@'],
output: ['@BASENAME@.c', '@BASENAME@.h'])

# Include directory and a dependency file.
# The generator writes the dep file automatically
# next to the output file.
g = generator(gen_exe,
arguments: ['@INPUT@', '-o', '@OUTPUT@', '--dep'],
depfile: '@OUTPUT@.d',
output: '@BASENAME@.c')

```

19.24 `get_option(name, ...)` → various

Returns the value of the given option. These can be either Meson global options such as `prefix` or current project's options. Accessing options of other projects is prohibited.

The type of the return value depends on the option. It can be a string, integer, array, or a feature object.

Positional arguments

name is the option to look up.

Usage examples

```

# On Unix machines this is typically '/usr'
p = get_option('prefix')

# Building an absolute path to the library directory.
ldir = get_option('prefix') / get_option('libdir')

```

19.25 `get_variable(name, fallback)` → object

Get the value of the build file variable with the given name.

Positional arguments

name is a string specifying the variable to look up.

fallback is an optional value that is returned if the specified variable did not exist. Requesting a nonexistent variable without specifying a fallback is a hard error.

251

Usage examples

```
var = 'text'

# Identical to var2 = var1
var2 = get_variable('var1')

# var3 gets the value 'default'
var3 = get_variable('nonexisting', 'default')
```

19.26 import(name) → module object

Loads a Meson extension module and returns it. Extension modules include additional helper functionality for frameworks and tools that is not the core language.

Positional arguments

name is a string with the name of the module to import.

Usage examples

```
# Create a Python extension module
python_mod = import('python3')
py3_dep = dependency('python3')
python_mod.extension_module('baz', 'baz.c',
    dependencies: py3_dep)
```

19.27 include_directories(dirs, ...) → include directories object

Creates an include directories object containing the given directory names. The command verifies that all given directory names exist and errors out if they

don't. The return value can be used in any other subdirectory, and Meson will automatically make paths work correctly.

252

When the return value is used in a build target, Meson will add both the source and build directories to the target's include path.

Positional arguments

dirs contains one or more strings to place in the include dir object. Relative paths are interpreted relative to the current source dir.

Keyword arguments

is_system is a boolean specifying if the include directive should be defined as a *system directory*. This is used to make compilers not print warnings that come from files in these directories. Some toolchains do not support the concept of system directories and on those this keyword argument is ignored.

Usage examples

```
# Store the directory
inc = include_directories('include')

# Use the result in a different directory.
executable(..., include_directories: inc)
```

19.28 `install_data(files, ...)` → none

Installs individual files.

Positional arguments

files are the files to be installed.

Keyword arguments

install_dir is a string defining the target directory for the files. If not defined the default value is `datadir/project name` where `datadir` is the value of the Meson option with the same name.

install_mode behaves identical to the same keyword in a build target as described on page 237.

rename is an array specifying if and how the files should be renamed upon install. Each item can be either **false** meaning the file won't be renamed or a string specifying the installed name.

253

Usage examples

```
# Prefix has been set to '/usr'
# Datadir has been set to 'share'
# Project name has been set to 'example'

# Installs to /usr/share/example/texture.png
install_data('texture.png')

# Installs to /usr/share/example/installed.png
install_data('in_sourcetree.png',
             rename: 'installed.png')

# Installs to /usr/share/example/texture.png
# and to /usr/share/example/sub/installed.png
install_data('texture.png', 'in_sourcetree.png',
             rename: [false, 'sub/installed.png'])
```

19.29 install_headers(files, ...) → none

Installs files in the specified header directory as defined by the option **headerdir**.

Positional arguments

headers list of files to be installed.

Keyword arguments

install_dir overrides the default install directory. This path is interpreted relative to prefix, not headerdir.

install_mode behaves identical to the same keyword in a build target as described on page 237.

subdir specifies a subdir under the include dir path where files should be installed to. If both **subdir** and **install_dir** keywords are specified, the latter has precedence.

Usage examples

254

```
# Prefix has been set to '/usr'
# Headerdir has been set to 'include'

# Install a single header to
# /usr/include
install_header('myheader.h')

# Path segments in source files are dropped.
# Both these files go to /usr/include
install_header('header1.h', 'sub/header2.h')

# This header goes to /usr/include/myproj
install_header('myheader.h', subdir: 'myproj')

# This header goes to /usr/foo/bar.
install_header('myheader.h', install_dir: 'foo/bar')
```

19.30 `install_man(files, ...)` → none

Install man pages to the correct system directory. Man pages have their own file system layout requirements. Meson will do all that work automatically, assuming that the files have been named in the standard way for man pages.

The files must be in the manpage format. Meson does not do any validation on file contents, but instead assumes them to be correct.

Positional arguments

files are the man page files to be installed.

Keyword arguments

install_dir can be used to override the directory to install to.

install_mode behaves identical to the same keyword in a build target as described on page 237.

Usage examples

```
# Man dir on this system is /usr/share/man

# Installs to /usr/share/man/man1
install_man('myprog.1')
```

19.31 `install_subdir(subdir, ...)` → none

Installs the subdirectory and all its contents transitively.

Positional arguments

subdir is the subdirectory to be installed.

Keyword arguments

exclude_files is a list of files, relative the the subdirectory, that should not be installed.

exclude_directories is a list of directory trees, relative to the subdirectory, that should not be installed.

install_dir is a string specifying the target directory.

install_mode behaves identical to the same keyword in a build target as described on page 237.

strip_directory is a boolean that can be used to strip the leading directory. Another way of looking at this is that the value of **true** will install the *contents* of the specified directory rather than the directory itself.

Usage examples

```
# Assume we have the following tree structure:
#
# topdir
#   sub1
#     sub2
#       sub2file.dat
#     sub1file.dat
#   topfile.dat

# Install result is:
# usr
#   share
```

256

```
#      prog
#      topdir
#      sub1
#      sub2
#          sub2file.dat
#          sub1file.dat
#      topfile.dat
install_subdir('topdir', install_dir: 'share/prog')

# Install result is:
# usr
#   share
#     prog
#       sub1
#       sub2
#         sub2file.dat
#         sub1file.dat
#       topfile.dat
install_subdir('topdir',
               install_dir: 'share/prog',
               strip_directory: true)

# Install result is:
# usr
#   share
#     prog
#       sub2
#         sub2file.dat
install_subdir('topdir/sub1/sub2',
               install_dir: 'share/prog')
```

19.32 `is_disabler(variable)` → boolean

Returns **true** if the variable given as the argument is a disabler object and **false** otherwise.

19.33 `is_variable(variable_name)` → `boolean`

Returns `true` if the variable with the name given as the argument exists and `false` if not.

257

Positional arguments

variable_name is a string specifying the variable to look up.

Usage examples

```
var = 42

# Returns true
is_variable('var')

# Returns false
is_variable('nonexisting')
```

19.34 `jar(name, sources, ...)` → `none`

Defines a Java jar target. This is its own object type because jars are special in that they can be both libraries and executables at the same time. Calling this function without enabling Java is a hard error.

Positional arguments

name is the name of the jar file (without the `.jar` suffix).

sources are source files to compile in this target.

Keyword arguments

main_class defines the Java main class, that is, the class that holds the main function. The Java run-time system needs this information in order to be able to run this jar file as an executable. The class must be specified in the Java class notation i.e. as `com.mesonbuild.ClassName`.

In addition this function accepts all the same arguments as `build_target`.

Usage examples

258

```
# The simple case
jar('sample', 'com/mesonbuild/prog/Helper.java')

# An executable jar.
# Can be run with "java -jar prog.jar"
jar('prog', 'com/mesonbuild/prog/SampleProg.java',
    main_class: 'com.mesonbuild.SampleProg')
```

19.35 `join_paths(paths)` → `none`

This function is deprecated and should not be used in new code. Paths should be joined with the `/` operator instead.

19.36 `library(name, sources, ...)` → `library object`

Builds a library target that can be either a shared, static, or both depending on the value of the `default_library` option.

All arguments behave in the same way as in other library functions. See `static_library` and `shared_library` for further information.

19.37 `message(text)` → `none`

Prints the given string to standard out.

Positional arguments

text contains one or more strings to print. Multiple strings are separated by a space.

Usage examples

```
# Prints out the following:
# Message: This is a message.
message('This is a message.')
```

19.38 `run_command(command, args, ...)` → `run_result`

Positional arguments

259

command is the command to run, either a string, a file type object, compiler object or the result of `find_program`. Note that this may not be an **executable** as the command is invoked immediately, that is, before any compilations have been done.

args are the command line arguments to use. They should be in an array like `['-foo', '-bar']` rather than a single string like `'foo -bar'`.

19.39 `run_target(name)` → `none`

Creates a new top level target that runs the specified command when invoked. This target is meant to give names to standalone commands that need to be run every now and then, such as invoking a source code formatter on the entire project.

The working directory is unspecified and may be different on different back-ends.

Positional arguments

name is the name given to the target.

Keyword arguments

command is an array specifying the command to run. The first argument is a program and the rest are the arguments given to it. These can be either strings or build targets, which will be automatically expanded to point to the output files. Meson will also add suitable target dependencies so that those targets are guaranteed to be up to date before the run target command is executed.

depends lists additional targets that need to be up to date before this command can be run even though they are not listed on the command line.

Usage examples

```
exe = executable(...)
lib = library(...)
formatter = find_program('code-formatter')
```

260

```
# Reformat all source code
run_target('reformat',
    command: [formatter, '--dir', meson.current_source_dir()])

# Both exe and lib are built before running the command.
run_target('something',
    command: [exe, '--do-something-with', lib])

# The program searches for the library on its
# own. We need to tell Meson that the library
# must be built before running this command.
run_target('something',
    command: [exe, '--magic-glob'],
    depends: lib)
```

19.40 `set_variable(name, value)` → none

Sets the variable with the given name to the given value.

Positional arguments

name is a string specifying the variable to set.

value is the value to set.

Usage examples

```
# This is identical to
# x = 'value'
set_variable('x', 'value')
```

19.41 `shared_library(name, sources, ...)` → shared library object

Creates a shared library.

Positional arguments

name is the name of the library. This name should not have a “lib” prefix, it will be added automatically on platforms that require it.

sources is a list of sources to use in this target.

Keyword arguments

darwin_versions can be used to specify macOS **compatibility_version** and **current_version** properties. If the argument has only one value, it will be used for both properties. If the argument is an array with two elements, they will be used for the two properties respectively. If this keyword is omitted, the value of **soversion** is used instead if it exists.

soversion is a string or an integer specifying the soversion of this library. The value should be a single number and will be automatically mapped to the appropriate platform specific behaviour (these vary widely). If not specified, the first digit of the **version** keyword argument is used instead if it exists.

version is a string specifying the version of this library. This string should consist of three digits separated by periods, such as 1.2.3. If not specified the **soversion** argument is used instead.

vs_module_defs is a file defining how symbols should be exported from the current library. Can be a string, File object or the output of custom target and it must be in the format understood by the Visual Studio compiler. Ignored on other compilers.

In addition this function accepts all keyword arguments of a **build_target**.

Usage examples

```
# Basic library
lib = shared_library('foobar', 'foobar.c')

# Versioned library
lib = shared_library('foobar', 'foobar.c',
                    version: '1.2.3',
                    soversion: '1')
```

19.42 shared_module(name, sources, ...)

→ shared module object

Builds a shared module. The difference between shared libraries and modules is that the latter are not linked against. They are only opened at runtime with **dlopen** or an equivalent function.

All arguments to this function are the same as for **shared_library**.

19.43 `static_library(name, sources, ...)` → static library object

262

Builds a static library with the given name.

Positional arguments

name is the name of the library. This name should not have a “lib” prefix, it will be added automatically on platforms that require it.

sources is a list of sources to use in this target.

Keyword arguments

pic is a boolean specifying whether the code should be built as position independent code. This is required if the library will be linked to either a shared library or a position-independent executable. If unset, the value of option `b_staticpic` will be used instead.

In addition this function accepts all keyword arguments of a `build_target`.

Usage examples

```
# Basic library
lib = static_library('foobar', 'foobar.c')

# This static library does in a shared library so
# build it with PIC unconditionally.
stlib = static_library(...,
                       pic: true)
shared_library(...,
               link_with: stlib)
```

19.44 `subdir(dirname, ...)` → none

Recurses to the specified subdirectory and executes the `meson.build` file there. Trying to enter the same directory twice is a hard error as is trying to go outside the source tree or cross a subproject boundary.

Positional arguments

dirname is a string specifying the directory to enter. Can contain multiple path segments.

Keyword arguments

is_found takes one or more dependency objects and only goes in the subdirectory if all dependencies are found.

Usage examples

```
subdir('src')

# Directory is not entered
subdir('tests', if_found: unfound_dependency)
```

19.45 `subdir_done()` → none

Stops processing this file and returns to the parent directory.

Usage examples

```
subdir_done()
message('This message will never be printed.')
```

19.46 `subproject(name, ...)` → subproject object

Executes the specified subproject. The project must be in the `subprojects` directory in the master project's root. A subproject can be invoked multiple times but it will only be executed once. Subsequent subprocess invocations return the same value as the first one.

Usually subprojects are not entered directly, but instead they are used indirectly via the `dependency` function's `fallback` keyword argument.

Positional arguments

name is a string specifying the name of the subproject.

264

Keyword arguments

default_options can be used to override the project's default options. Its behaviour is identical to the same keyword argument in **dependency**.

version specifies version requirements for the subproject. Its behaviour is identical to the same keyword in **dependency**.

Usage examples

```
# Basic usage
sp = subproject('somelib')

# Version requirements
sp = subproject('somelib',
               version: ['>=1.0.0', '<2.0.0'])
```

19.47 summary(values, ...) → none

Calling this function causes a piece of summary data to be printed at the end of the configuration run, just before Meson exits. This is most commonly used to print configuration items in a coordinated fashion. Each summary item consists of three pieces, a *key*, a *value* and an optional *section*. The values will be printed grouped first by subprojects and then by sections.

Positional arguments

values specifies the value(s) to print. It takes either two arguments which are the key and the value or a dictionary of key–value pairs. The behaviour of the latter is the same as calling this function for each individual entry in the dictionary. Keys must be strings. Values can be either a string, integer, a boolean or an array of same.

Keyword arguments

bool_yn if set to **true**, prints boolean values as **yes** and **no** rather than **true** and **false**. Defaults to **false**

list_sep is a string that defines how the elements of an array should be separated. If omitted, the values are separated by a newline.

section lists the section this message should be printed under. If unset, the summary is printed in the top level “project” section.

Usage examples

```
# The following two calls are identical
summary('key', 'value')
d = {'key': 'value'}
summary(d)

# The contents of arrays are prettyprinted
summary('plugins', ['one', 'two', 'three'])

summary('Printer support', True,
        section: 'features',
        bool_yn: true)

# Comma separated values.
summary('choices',
        ['one', 'two', 'three'],
        list_sep: ', ') # Note the space.
```

265

19.48 test(name, executable, ...) → none

Defines a test, which is an executable and all related state needed to run it. The outcome of a test depends on the return value of the program. Zero is success, 77 means skipped and every other value means failure. A skipped test is one that could not be run, usually because some runtime information is not available, but that this should not be considered a failure.

Positional arguments

name is the name of the test. It is printed in logs and can be passed to **meson test** to run individual tests.

executable is the program to run. Can be the return value of **executable**, **custom_target** or **find_program**.

Keyword arguments

args contains the command line arguments to pass to the program. The most common ones are strings, but you can also pass Meson objects such as build targets. They will automatically be converted to strings that point to the results.

266

- depends** lists those targets that are needed to run this test, even though they are not used as command line arguments. Meson ensures that these targets are built and up to date before running this test.
- env** lists the environment variables that should be set. This is typically an environment object, but can also be an array of strings in the format ['key1=value1', 'key2=value2'].
- is_parallel** is a boolean specifying whether the test can be run in parallel with other tests. If set to **false** the test runner will ensure that no other test is running at the same time.
- priority** is an integer listing the priority level of this test. This value can be positive or negative with the default value of 0. The test runner arranges tests to be run in decreasing order of priority.
- protocol** specifies the type of the test. The default value is **exitcode** but it can also be **gtest** or **tap** for better integration with Google Test or the Test Anything Protocol, respectively.
- should_fail** inverts the success logic. If set to **true** the test is considered successful if it returns a non-zero return value and failed if it returns zero. A return value of 77 is always considered as skip.
- suite** has one or more strings defining the test suites for this test. The Meson test runner can then be told to run, or not run, certain suites with the **--suite** and **--no-suite** command line arguments.
- timeout** is a number specifying how long in seconds this test is allowed to run. A test that exceeds this amount is killed and the test is marked as failed. The default value is 30 seconds.
- workdir** is an absolute path to a directory that should be used as the working directory of the test. The default working directory is unspecified.

Usage examples

```
exe = executable(...)
lib = library(...)

# Passing the library as an argument.
# Meson will convert the lib object into a string
# pointing to the lib file automatically.
test('args', exe,
     args: lib)

# Run the test in a different directory.
test('in_tmp', exe,
```

```

    workdir: '/tmp')

# Set environment variables
e = environment()
e.set('NAME', 'VALUE')
test('envvar', exe
     env: e)

# Tag test as slow:
test('slow', exe,
     suites: 'slow')
# Run all tests except those tagged as slow:
meson test --no-suite slow

```

19.49 vcs_tag(...) → file object

This function takes a user defined template file and generates a new file that contains the version control commit ID. The generated file is guaranteed to always be up to date. If the current commit id changes, such as via doing a new commit, the next build will contain the new commit id.

Keyword arguments

command is a command array in the same format as in `custom_target`. If specified the file will be generated with this command rather than string substitution.

fallback is the value if the project is built outside revision control, e.g. from a release tarball. If omitted, defaults to the current project version.

input is the input template file.

output is the output file name.

replace_string is the string that should be replace. The default value is `@VCS_TAG@`.

Usage examples

```

# contents of template file are:
# #define VCSTAG "@VCS_TAG@"

vcs_header = vcs_tag(input: 'vcs_version.h.in',
                    output: 'vcs_version.h')
executable(..., vcs_header, ...)

```


Chapter 20

Module reference

Modules in Meson are kind of like the standard library in programming languages. They provide functionality that is useful, but not fundamental enough to be part of the core language. All module functionality is used in the same way.

```
my_mod = import('modulename')
my_mod.do_something(...)
```

The `import` command is used to activate the module, which is stored in a variable. Calling methods on this variable allows you to use the functionality. In addition to modules listed here Meson has several unstable methods

20.1 cmake module

The `cmake` module can be used to generate CMake dependency information files on install. These dependencies are called *packages* in CMake. This module consists of two methods which have the same names and behaviour as CMake functions with the same names. Extensive documentation on their usage can be found on CMake's reference documentation.

20.1.1 `write_basic_package_version_file(...)` → none

Generates and installs a CMake file that specifies a package and its version.

Keyword arguments

name is the name of the CMake package to create.

version is the version of the package as a string.

270

compatibility specifies the dependency compatibility type. It can be set to `AnyNewerVersion` (the default), `SameMajorVersion`, `SameMinorVersion` or `ExactVersion`

install_dir is the directory to install to. If not set, defaults to the value `get_option('libdir') / 'cmake' / name`.

20.1.2 `configure_package_config_file(...)` → `none`

Generates and installs a file that specifies a CMake module. The previous method defines only the version, this method generates all the information needed to actually use the module. This method behaves similarly to the `configure_file` Meson function. It generates the output file from an input template and a configuration data object.

Keyword arguments

name is the name of the package as a string.

input is a template file for the package. The format should be the same as in CMake.

install_dir specifies the directory to install to. If not set, defaults to the value `get_option('libdir') / 'cmake' / name`

configuration is a configuration data object with the values that will be written in the output file.

Usage examples

```
# In CMake you need to generate both a version and
# a config file to get a working dependency module.
cmake_mod = import('cmake')
conf = configuration_data()
conf.set(...)
cmake_mod.write_basic_package_version_file(name: 'myProject',
                                           version: '1.0.0')

cmake_mod.configure_package_config_file(
    input: myProjectConfig.cmake.in,
    name: 'myProject',
    configuration: conf,
)
```

20.2 dlang module

This method provides helper functionality for programs and libraries written in the D programming language.

271

20.2.1 generate_dub_file(project_name, source_dir, ...)
→ none

Dub is the D language's package manager and dependency provider system. This function generates a file needed to integrate a Meson project with Dub.

Positional arguments

project_name is the Dub name for this project.

source_dir points to the directory holding the D source for this project.

Keyword arguments

This method accepts arbitrary keyword arguments with string values. They are written in the Dub JSON file as-is. If a file for the given project already exists, Meson will only overwrite the key values listed in keyword arguments. All other values in the JSON will be preserved.

Usage examples

```
dlang.generate_dub_file('myprogram', meson.source_root(),
                        # These two are required for packages
                        # submitted to Dub package registry.
                        description: 'Sample program',
                        license: 'MIT')
```

20.3 fs (filesystem) module

The filesystem module contains functionality to inspect the contents of the file system and to manipulate paths. Methods of the latter type do not touch the file system, they operate purely on strings.

All methods treat non-absolute paths as being relative to the directory where the current `meson.build` file resides.

Meson exposes the underlying file system semantics directly rather than trying to define cross platform behaviour on top of it. This means that some methods may work differently on different platforms. This also means that all

methods follow symlinks transparently unless otherwise specified. Trying to access non-existing filesystem entries always leads to a fatal error.

272

20.3.1 `exists(entry_name)` → boolean

Returns `true` if the argument string points to an entity on the file system. It can be either a file, directory, Unix device or something else. This function may return `false` even if the given file actually exists, for example because the current user does not have read access to the file.

Positional arguments

entry_name is a string holding the file name to check, for example `d/file.txt`.

20.3.2 `hash(file_name, hash_type)` → string

Calculates a checksum of the given file's contents. The return value is a string in the common human-readable "hex digest" format.

Positional arguments

file_name is the file name to process.

hash_type is the hash algorithm to use, can be one of `md5`, `sha1`, `sha224`, `sha256`, `sha384` or `sha512`.

Usage example

```
# the input file contains the text "abc123"
h = fs.hash('input_file.txt', 'sha1')
# Value of h is:
# 6367c48dd193d56ea7b0baad25b19455e529f5ee
```

20.3.3 `is_dir(dir_name)` → boolean

Identical to `exists` but returns `true` only if the thing pointed to is a directory.

20.3.4 `is_file(file_name)` → boolean

Identical to `exists` but returns `true` only if the thing pointed to is a regular file.

20.3.5 `is_samepath(file1, file2) → boolean`

Returns `true` if the two argument strings point to the same file. This can be either because one is a symlink to another or because the paths resolve to the same entry.

Usage example

```
filename1 = 'file.txt'
filename2 = './file.txt'
filename3 = 'symlink_to_file.txt'

fs.is_samepath(filename1, filename2) # returns true
fs.is_samepath(filename1, filename3) # returns true
```

20.3.6 `is_symlink(file_name) → boolean`

Identical to `exists` but returns `true` only if the thing pointed to is a symlink.

20.3.7 `name(file_name) → string`

Returns the last path segment (the “file name”) of the given path string.

Usage examples

```
fs.file_name('dir/file').name()      # returns 'file'
fs.file_name('dir1/dir2/file').name() # returns 'file'
```

20.3.8 `parent(file_name) → string`

Returns the “parent directory” of the given filename string.

Usage examples

```
fs.parent('dir/file')      # dir
fs.parent('dir1/dir2/file') # dir1/dir2
```

20.3.9 `replace_suffix(filename, new_suffix) → string`

This method can be used to manipulate the suffix (or file extension) of the given file name. Note that this method does not touch the file system, it only does string replacements. The file suffix is defined to be everything following the last period in the file name. Thus the suffix for `name.suf1.suf2` is `.suf2` and for `dir.1/fname` the suffix is the empty string.

Usage example

```
fs.replace_suffix('file.suf1.suf2', '') # file.suf1
fs.replace_suffix('file.suf', '.bak')   # file.bak
fs.replace_suffix('file', '.suf')       # file.suf
```

20.3.10 `size(file_name) → integer`

Returns the size of the given file in bytes.

20.4 `gnome` module

The Gnome module provides helper functionality for tools provided by the GNOME development platform. Many of these are fundamental pieces of infrastructure used on all Linux systems.

Many methods and especially their arguments in this module map directly to the underlying concepts of the tools and frameworks used. An in-depth explanation could take a book all its own and is thus out of focus for this manual. This section aims to give a rough understanding of the various parts, but for details the reader is encouraged to look up the reference documentation as provided by GNOME.

20.4.1 `compile_resources(name, resource_file, ...)` `→ [c_source, h_source] or [resource_bundle]`

The GResource mechanism is a way to embed binary data directly in executables. Its input is a file written in XML specifying which files should be embedded. The resource compiler reads the file and converts the contents of the resources into plain C arrays that can be compiled in the application or into a special bundle file that can be loaded at runtime. The generation is done with the `glib-compile-resources` tool provided by GLib.

When generating sources, the return value of this function should be put in the source list of the target that uses said resources.

Positional arguments

name is the base name given to the resource file.

resource_file is a path pointing to the resource file to be processed. By convention the file's name is `projname.gresources.xml`.

Keyword arguments

c_name specifies the name prefix to be used for entries when generating source code.

dependencies is a list of extra targets that need to be built before this target can be run. Usually because the output of those targets is put in this resource setup.

export a boolean telling whether the symbols in the generated source code should be exported. Defaults to **false**.

extra_args is an array of strings that will be passed to the resource compiler unaltered.

gresource_bundle will, if set to **true**, generate a resource bundle instead of source code. Note that this option changes the type of the return value. Defaults to **false**.

install is a boolean specifying whether the generated resource bundle should be installed or not.

install_dir specifies the directory where installed artifacts should go.

install_header causes, when set to **true**, the generated header to be installed.

source_dir is a list of directories where the resource compiler should look for binary files.

Usage examples

```
s_res = gnome.compile_resources(  
    'sample', 'sample.gresource.xml',  
    source_dir: 'textures', # look up files in this subdir  
    c_name: 'smpl'  
)  
  
# The return value must be used in the target  
# that uses the resources.  
executable(..., s_res, ...)
```

20.4.2 `compile_schemas(...)` → `none`

In the context of GNOME a *schema* defines the settings an application has. These schemas are defined in XML files that need to be compiled into a binary form and installed. This is done with the `glib-compile-schemas` program. This program takes no input, all schema files must reside in the current source directory and must have the extension `.gschema.xml`.

Keyword arguments

build_by_default if set to `true` (which is the default) causes the schemas to be built during normal compilation. Otherwise they are only built on install.

depend_files is a list of files that should be listed as schema sources, meaning that changing them will cause the schema file to be recompiled.

20.4.3 `gdbus_codegen(name, ...)` → [`c_source`, `h_source`, `docbook_target`]

D-Bus is a widely used RPC (remote procedure call) mechanism. GDBus is GNOME's client and server implementation of the protocol. Like most RPC mechanism, D-Bus works by defining the interface with a custom description language. This description is then converted to source code with a compiler tool. For GDBus this tool is called `gdbus-codegen`.

The return value is an array with the source and header targets. If the `docbook` keyword argument is set to `true` the return value will have the docbook target as its third element.

Positional arguments

name is a name for this compilation target.

Keyword arguments

annotation is an array of arrays where each entry defines one annotation with three strings. Each entry has the following layout: [`ELEMENT`, `KEY`, `VALUE`].

autocleanup specifies whether to generate autocleanup code, permitted values are `none`, `objects` and `all`.

build_by_default if set to `true` will build the targets during a regular build. The default is `false`.

docbook is a string specifying the file name prefix for generated docbook documentation. If not defined, docbook data is not generated.

install_dir the directory where the header or bundle will be installed.
install_header if `true` installs the header file. Default value is `false`.
interface_prefix is a string that is prepended to all interface names.
namespace is a string specifying the namespace in the generated C source, should be in either `CamelCase` or `Ugly_Case`.
extra_args is an array of strings that will be passed to the command line of the source generator.
object_manager if `true` generates object manager classes. Default value is `false`.
sources list of XML files specifying the D-Bus services.

20.4.4 `generate_gir(targets, ...)` → `[gir_target, typelib_target]`

This method generates GObject introspection files for the given sources and targets. This format is a machine readable description of the API and ABI, which can be used to e.g. automatically generate bindings for many languages such as Python. The generation is implemented by calling the `g-ir-scanner` and `g-ir-compiler` tools provided by GLib.

Generating GIR data requires both the source and the built artefact (library or executable). It can not be created merely by scanning source code. Introspection data is split in two files: a gir file and a typelib file. Thus the return value is an array so the outputs can be used individually.

Positional arguments

targets are build targets for which the GIR data will be generated. These are targets built as part of the current project. They can either be a single executable target or multiple library targets.

Keyword arguments

dependencies extra targets that must be built before introspection can be done.

extra_args is an array of strings that will be passed to the introspection scanner invocation unaltered.

export_packages a list of strings of extra package names that this GIR file exports.

header specifies the main header needed to use this library.

identifier_prefix specifies the prefix of this object.

include_directories an array of directories used to look up headers.

includes list of GIR file names or a `gir_target` used to search includes.

278

install if **true**, the introspection files are installed.

install_dir_gir is the directory where the gir file will be installed. If unset, defaults to **share/gir-1.0**.

install_dir_typelib is the directory where the typelib file will be installed. If unset, defaults to **share/girepository-1.0**.

namespace is a string specifying the namespace of this unit.

nsversion is a string specifying the namespace version.

link_with is a list of built libraries to link against.

sources contain the source files that should be scanned for GIR data.

symbol_prefix is an array of one or more symbol prefix strings to be used in the gir object.

20.4.5 `generate_vapi(library_name, ...)` → `dependency`

Generates a VAPI file from a GIR file. Vapi files are library description files used by the Vala programming language.

Positional arguments

library_name is name of the generated library.

Keyword arguments

gir_dirs is a list of directories used to look up gir files.

install if **true**, installs the file. Default value is **false**.

install_dir is the directory where the generated VAPI files will be installed.

metadata_dirs is a list of directories where extra metadata files should be looked up in.

packages is an array of strings listing the VAPI packages this package depends on.

sources lists the GIR sources to use. The values can be either strings or GIR targets from `generate_gir_targets`.

vapi_dirs is a list of strings specifying directories to use for looking up VAPI files.

20.4.6 `genmarshal(basename, ...)` → `[c_source, h_source]`

This method generates *marshalling code* which is needed to work with the GObject callback mechanism. The code is generated using the `glib-genmarshal` tool provided by GLib.

Positional arguments

basename is the basename used for the output files.

279

Keyword arguments

extra_args is an array of strings that will be passed to the generator program unaltered.

install_header if **true**, install the generated header. Defaults to **false**.

install_dir the directory to install the header to. If **install_header** is **true** this argument must be defined..

nostdinc if **true** do not use the default marshallers. The default value is **false**.

internal if set to **true** causes the generated marshallers to be marked as internal. Default value is **false**.

prefix is a string specifying the prefix to add to all symbols.

skip_source , if **true**, tells the generator to skip source locations in generated comments.

20.4.7 gtdoc(module_name, ...) → none

This method generates and installs documentation with the gtdoc tool. It can gather documentation both by scanning source code for documentation comments as well as introspecting existing libraries for GObject class data.

Positional arguments

module_name is the name of the Gtk-Doc module.

Keyword arguments

c_args is an array of extra compiler arguments that GTK-doc should use when compiling code.

content_files are files that need to be copied to the build tree to build the documentation. This file layout is required by Gtk-Doc.

dependencies are targets that must be built before the documentation can be generated. Typically includes libraries that will be introspected.

gobject_typesfile specifies which GObject types file should be used for GObject introspection.

include_directories is an array of extra directories that should be passed to `gtkdoc-scangobj`.

html_assets is an array of files that should be copied to the documentation's asset directory.

install installs the generated documentation if **true**.

280

install_dir is a directory where the docs should be installed. If the given path is not absolute it is interpreted relative to `share/gtk-doc/html`. The default value is the module name.

main_sgml specifies the name of the main SGML file.

main_xml specifies the name of the main XML file. It is illegal to specify both this and **main_sgml**.

module_version is a string specifying the module's version.

src_dir has one or more include directories objects specifying where Gtk-Doc should scan for source file comments.

In addition this method accepts the following keyword arguments that are mapped directly to the respective Gtk-Doc tools' command line arguments: `ignore_headers`, `fixxref_args`, `mkdb_args`, `namespace` and `scan_args`

20.4.8 `gtkdoc_html_dir(name) → string`

When given a gtkdoc module name **name** returns a string specifying the path where said module's HTML files should be installed. This method is mostly used to generate paths to use in **install_dir** keyword arguments.

20.4.9 `mkenums(basename, ...) → [c_source, h_source]`

This method converts a plain C enum declaration into a richer GObject type. This is done with the `glib-mkenums` program provided by GLib. Most projects should use the `mkenums_simple` method instead.

This method takes two classes of keyword arguments. The first ones map directly to keyword arguments of other generators in this module and include **install_header**, **install** and **sources**. The second class maps directly to command line arguments accepted by the compiler and include **c_template**, **h_template**, **comments**, **identifier_prefix**, **symbol_prefix**, **eprod**, **fhead**, **fprod**, **ftail**, **vhead** and **vtail**.

20.4.10 `mkenums_simple(basename, ...)` `→ [c_source, h_source]`

This is a simpler version of **mkenums** for projects that follow GLib's recommended enum layout.

Positional arguments

basename specifies the base portion of output file names.

Keyword arguments

body_prefix is an extra string to be written at the top of the generated source file.

decorator is a string that will be added before function declarations. This can then be redefined with a preprocessor macro.

function_prefix is a string that will be prepended to every function name.

header_prefix is an extra string written at the top of the generated header file.

identifier_prefix is a string that will be prepended to every identifier name.

install_header if **true** installs the header. Defaults to **false**.

install_dir specifies the directory where the header gets installed.

sources is the list of sources to scan for enum declarations.

symbol_prefix is a string that will be prepended to every symbol name.

20.4.11 `yelp(projectname, ...)` → `none`

Uses GNOME's Yelp program to install help documentation. It uses the command line tools `itstool`, `msgmerge` and `msgfmt`.

All keyword arguments that refer to files assume that they are in a subdirectory called **C** in the current source directory. Meson will automatically set up two run targets that can be used to update translations. For a project called `myproj`, the target names are `help-myproj-update-po` and `help-myproj-pot`.

Positional arguments

projectname is the Yelp project ID for the current set of documentation.

Keyword arguments

languages is a list of strings defining which languages to use for translations.

media is a list of media files used in the documentation.

sources is a list of documentation source pages.

symlink_media is a boolean specifying that the media files should be symlinked rather than copied on install. The default value is **true**.

20.5 hotdoc module

HotDoc is a program for generating documentation pages from source code comments.

20.5.1 `generate_doc(project_name, ...)` → hotdoc object

Generates documentation for the given project. A hotdoc object is a custom target with an extra method `config_path`. It returns a string pointing to the generated configuration file.

Positional arguments

project_name is a string specifying the HotDoc project name for this documentation set.

Keyword arguments

sitemap is a string or a file pointing to the HotDoc sitemap file, which specifies the layout of generated documentation pages.

index is a string or file pointing to the index file, that is, the “front page” of the documentation.

dependencies is an array of targets that must be fully built before documentation generation can begin.

subprojects is an array of one or more hotdoc objects that will be nested inside this project.

In addition any argument accepted by the `hotdoc` program can be used as a keyword argument by replacing dashes with underscores. For example the argument `--c-sources` could be specified with the keyword argument `c_sources`.

20.6 i18n module

This module provides helper functionality for internationalisation and localisation of programs, such as translating application text to multiple languages.

20.6.1 `gettext(name, ...)` → none

Gettext is the most popular translation framework on unixlike operating systems. This method builds and installs all files needed by gettext localisations.

Positional arguments

name is the Gettext module string.

Keyword arguments

args is an array of strings that will be passed to the `xgettext` program when generating the translation `pot` file.

data_dirs is an array of strings specifying directories that the `gettext` program should use to look up `its` files.

languages lists the languages that should be generated. This argument is deprecated, projects should use a `LINGUAS` file instead as described in `Gettext` documentation.

preset selects a configuration to use. The configuration consists of command line arguments that will be used. Currently the only supported value is `glib`, which should be used for all projects using `GLib`.

install tells whether generated localisation files should be installed. Defaults to `true`.

install_dir is the directory the localisation files should be installed to. Defaults to the value of the option `localedir`.

Extra targets

This method also creates three run targets that can be executed to update translation files. The first one is `<module>-pot`, which rescans all source files for translatable strings and adds them to the `pot` file. The second is `<module>-update-po` which updates `po` files that will be handed off to translators.

Note that unlike almost every other Meson target, running these changes the files in *source directories*. They are never run automatically, only when invoked manually by the user.

The final target is `<module>-gmo` which builds `gmo` files without installing them. They are binary files containing the translated texts that the `gettext` mechanism uses at runtime.

20.6.2 `merge_file(target_name, ...)`

→ custom target object

This method can be used to generate translated text files from a template file. The converted file must be in one of two formats: an XML file or a `.desktop` file.

This function accepts all the same keyword and positional arguments as a `custom_target`. In addition it supports the following keyword arguments.

Keyword arguments

data_dirs is identical to the same keyword argument in `gettext`.

po_dir is a path pointing to the directory holding the po files. By convention this directory should be at source root and called po.
type is either **xml** or **desktop**.

284

Usage examples

```
# Localise and install a .desktop file.
i18n.merge_file(
    input: 'test.desktop.in',
    output: 'test.desktop',
    type: 'desktop',
    po_dir: '../po',
    install: true,
    install_dir: get_option('datadir') / 'applications'
)
```

20.7 pkgconfig module

This module can be used to automatically generate pkg-config files on install. This makes it easy for other projects to use your code. Meson's **dependency** method primarily looks up dependencies with pkg-config files as do many other build systems. Pkg-config files are installed in the system's pkg-config directory, which on most systems is `/usr/lib/pkgconfig`. In Meson terms this is equivalent to `get_option('libdir') / 'pkgconfig'`.

A pkgconfig dependency typically defines some header include directories, libraries to link against and, less often, explicit compiler and linker arguments to use.

Pkg-config files also contain dependency information. This is needed if you, for example, want to link the dependency statically rather than dynamically, which is the default. The pkgconfig file generator will automatically add this dependency information. Defining which dependencies are added and how is quite complicated and, at the time of writing, not fully specified yet. It is likely that details of it have changed by the time you read this. If problems appear, the reader is suggested to look up the full specification on Meson's online documentation.

20.7.1 generate(...) → pkgconfig object

Creates and installs a pkg-config file as specified by the keyword arguments.

Keyword arguments

d_module_versions is a list of module version flags to use when compiling D language source files that use this dependency.

description is a short human readable text snippet describing the library.

extra_cflags is an array of strings of compiler flags that should be used with this library.

filebase is the name of the file to be generated without the `.pc` suffix. This will also be this dependency's Pkg-config module name.

install_dir is the directory to install the pc file to. Default value is determined as described above.

libraries is an array of library build targets that belong to this dependency. The values can contain plain strings too, they are assumed to define other libraries and will be put in the pc file's library list. The most common value is `-lm` that links against the C language's standard math library.

libraries_private is the same as **libraries** but the libraries are only used if the dependency is linked statically.

name is the human readable name of this library as reported by Pkg-config. Note that this is not the name you'd use to query dependency information. That is called the module name and is defined by the keyword argument **filebase**.

subdirs is a list of strings defining subdirectories of the global include directory that should be added to the header search path.

requires is an array of strings, dependencies or pkgonfig objects. These will be listed as dependencies of the current dependency.

requires_private is the same as **requires** but the dependency is marked as private.

url is a string with an URL pointing to the project's web site.

variables are an array of strings in the format **key=value**. These entries will be written as queryable variables in the pc file.

version is a string specifying the version number of this dependency. The default value is the current project version.

20.8 python module

The Python module makes it easy to build and install Python extension modules written in native code. The most common language is C but any language that can provide the same ABI can be used.

This module supports Python versions 2 and 3 with an identical interface.

20.8.1 `find_installation(name, ...)` → installation object

286

There are many different ways to install Python. This method introspects the system and tries to find the desired Python installation. The return object is documented in Section 18.13.

The **name** argument specifies how the installation should be looked up. If its value is `python3` or `python2`, Meson will try to find an installation of Python 3 or 2, respectively. There are no other requirements, any installation with the given major version is acceptable. If the argument is an absolute path to an executable, it is used as the installation. Finally if the argument is a plain name such as `python3.6`, it will be looked up internally with `find_program`.

Keyword arguments

required if set to `true` (the default) then failing to find the installation is a hard error.

disabler is a boolean. If set to `true` then failing to find the installation returns a disabler object instead. The default is `false`.

20.9 qt5 module

Qt is a popular C++ framework for creating GUI applications, though it can also be used for command line programs. It has some special requirements for build systems, because some input source files must be preprocessed with code generator tools before they can be compiled. This module aims to provide a simple user interface to the underlying functionality.

20.9.1 `preprocess(...)` → source object

There are three different code generator tools in Qt: `moc` for source code, `uic` for user interface files and `rcc` for general resources. This method processes all three types. The recommended way to use this method is to process all files of a single build target in one invocation of this method.

The return value is an opaque object. It should be passed to the build target definition in the list of sources.

Keyword arguments

moc_headers is a list of header files that need to be processed with `moc` (usually because they define `Q_OBJECTs`.)

moc_sources is a list of source files that need to be processed with `moc`.

moc_extra_arguments is a list of strings that will be used when invoking `moc`.

qresources is a list of resource files that need to be processed with `rcc`.

rcc_extra_arguments is a list of strings that will be used when invoking `rcc`.

uic_extra_arguments is a list of strings that will be used when invoking `uic`.

ui_files is a list of user interface description files that need to be processed with `uic`.

20.9.2 `compile_translations(...)` → `none`

This method is used to compile and install program translations with Qt's internationalisation tool `lrelease`.

Keyword arguments

ts_files is a list of files. These files are generated with Qt's `lupdate` tool that scans source files for strings to translate.

install is a boolean specifying whether the generated files should be installed.

install_dir is the directory to install the files to.

build_by_default is a boolean specifying whether the targets should be built by the default build command. Defaults to `false`, meaning the translated files are only generated on install.

20.10 qt4 module

The `qt4` module provides the same functionality as the `qt5` module, but for Qt version 4. All methods are identical in both modules.

20.11 rpm module

The RPM module provides an easy way to create a template `spec` file that is used for creating RPM packages. The module is used by adding this line in some build file:

```
import('rpm').generate_spec_template()
```

When Meson is configured it will write the template file in the build dir. This can be copied out and used. The line added above should be deleted as it is no longer required.

20.12 sourceset module

288

This module provides helper functionality for defining files and dependencies based on detected features and user configuration.

20.12.1 `source_set()` → `source_set` object

Creates a new source set object. All actual functionality of this module is accessed through these objects. They are fully described in Section 18.16.

20.13 windows module

This module provides helper functionality for building applications that run on the Windows operating system.

20.13.1 `compile_resources(rc_files, ...)` → `object`

In Windows many parts of an application, such as menus, icons and so on, are defined as *resources* in files that must be processed with a specific tool and compiled in the application.

The return value is an opaque object that should be put in the source list of a build target.

Positional arguments

`rc_files` are one or more resource files to compile.

Keyword arguments

`args` is an array of strings to pass as arguments to the resource compiler.

`depend_files` is a list of files the resource depends on. Changing any of these files causes the resource to be rebuilt. The resource compiler does not provide dependency information so this list of files can not be obtained automatically.

`depends` is a list of targets that this build depends on, even though it does not take them as command line arguments. Used when some of the resource files are generated with e.g. `custom_target`.

`include_directories` is a list of directories that should be used to look up resource files.

Part III:

The appendixes

Appendix A

Contributing to Meson

The Meson build system is not set in stone. By the time you read this book, it will have added new features that are not discussed in this book. Some of these features are created by the core Meson development team, but most are contributed by regular users of the program. As the health of any open source project depends on the number of contributions it gets, the Meson projects loves contributions and want to encourage as many people as possible to participate in its development. Not only is contributing fun and educational, it is also the most certain way of getting your own special feature in the system.

In this chapter we will give a rough outline on how to prepare, submit and polish your contribution and hopefully eventually get it merged in Meson. This process is not exclusive to Meson, most open source projects have a similar (and often times identical) process for contributions. Thus you can use this as a general guideline on how to contribute to open source projects, but do note that some projects may have unique requirements and processes so always check the correct procedure before contributing to a project.

In case you run into any problems in your code submission, you should seek assistance from the Meson development IRC channel where most of the developers hang out. Instructions on how to use IRC can be found with your favourite Internet search engine. The channel is called `#mesonbuild` and it is hosted on the Freenode network.

A.1 Checking out the code

The first thing you need to do is to get an up to date copy of Meson's Git repository. The development happens in Github, so you need to create a Github account. You also need to set up SSH key access, the details on that can be found in Github's documentation. Actual development is done using the Git version

control system. It is not a particularly intuitive or user friendly application, so getting started may be a bit intimidating.

First you need to create your own copy of the repository on your own Github account. In Github terminology this is called a *fork*. To create one, go to the Meson project page at <https://github.com/mesonbuild/meson> and look for a button at the top right corner that says “fork” and click it.

This should create a new copy and forward you to your own version of the code. Now you need to create your local copy of the repository so you can work on it. This is called a *clone*. Look for a green button that says *Clone or download* and click on it. It should open a text frame with a clone URL that should look like this:

```
git@github.com:<your_username>/meson.git
```

If the address instead starts with `https://`, click on the button that says *Use SSH*. The URL should change to the form listed above. If it does not, you need to set up SSH keys as mentioned earlier.

Next you need to open a command prompt. Change to a directory you want to check the code out in and issue the following command.

```
$ git clone git@github.com:<your_username>/meson.git
```

Once the command finishes you have your own copy of the repository, go inside the `meson` directory and run the following command:

```
$ git remote add upstream https://github.com/mesonbuild/meson.git
```

This tells Git that we want to track the original repository under the name `upstream`. This functionality will not be needed until much later in the process but it’s good practice to set it up immediately.

Meson development is heavily based on tests, so let’s run them all to verify that they pass.

```
$ ./run_tests.py
```

If you are on Windows you might need to manually use the Python command instead.

```
$ python run_tests.py
```

If any tests fail, it is probably because you are missing some build dependencies. Install them and rerun the tests until all tests pass. Do not skip this step! You can not know if your change works unless you can make all tests pass.

A.2 Creating the merge request

Now you need to implement the change. There are three main types of changes: documentation changes, bug fixes and new features. The Meson project has a requirement that changes in the code must come with tests. Bug fixes require regression tests to ensure that the bug is never inadvertently reintroduced. Features need tests to prove that they work as expected. Lack of tests is a sufficient reason to reject any code submission regardless of everything else.

For simplicity we are going to assume that the change consists of only one commit. Once the change has been written and tested, you can create a commit with your changes.

```
$ git commit -a -m 'Commit message here.'
```

If your commit fixes a bug in Meson's bug tracker you should add it to the commit message:

```
$ git commit -a -m 'Commit message here. Closes #1234.'
```

This causes the system to close the bug automatically when this commit is merged to Meson's master repository. Documentation only changes should look like this instead:

```
$ git commit -a -m 'Commit message here. [skip ci]'
```

In order to create a merge request the commit needs to be pushed to your Github account. By default Git pushes changes to that repository, so this is just a question of running:

```
$ git push
```

Then you can open your web browser and go to your repository's page and click the *New pull request* button. Github will show you the changes that will be in your pull request. If everything is ok, push the green *Create pull request* button.

You will get a page that allows you to write a message explaining your change. It is not needed for simple changes, but for complex changes you might want to write a few words. When done you can press this page's the *Create pull request* button.

A.3 Review and continuous integration

294

Meson has an extensive *continuous integration* system that runs the full test suite on multiple platforms and with many different toolchains. It is not uncommon for a pull request to work fine on the submitter's machine but fail when run on CI. If this happens then it is the submitter's responsibility to fix all test failures. Members of the development team will provide assistance as mentioned in the beginning of the chapter, but the final responsibility rests on the submitter as pull requests with failing tests will not be merged.

In addition to test failures you will most likely be faced with review comments from Meson developers. They range from pointing out typos to requesting major architectural changes to the commit. These commits are in no way meant to be personal but are done to ensure the reliability and especially long term maintainability of the project. The goal of the review process is to make the code submission itself better.

Regardless of whether you hit test failures or review comments you need to update your pull request. Once the code has been edited and working, the pull request can be updated. First you update the local commit.

```
$ git commit -a --amend
```

This tells Git not to create a new commit but instead add the current changes in the previous commit. Then we need to update the version on your Github page.

```
$ git push --force
```

The `--force` argument is needed because we change existing commits rather than add new ones. This will overwrite the remote repository, so you should be careful when force pushing.

After the push Github's automatic mechanisms take over, update the pull request with the new commit and re-run CI on the system. If all tests pass and maintainers are happy, the pull request will be merged. If not, there will be a new round of reviews and updates until the change is either merged or rejected.

A.4 Fixing merge conflicts

While a pull request is in review, other pull requests get merged to master. If a change that was merged changes the same lines of code as your pull request it creates a *merge conflict*. A pull request with a conflict can not be merged,

because there is no automatic way of reconciling these changes. This must be fixed manually by the pull request submitter.

There are two different ways of resolving these conflicts, *merging* and *rebasing*. Meson, like most open source projects, only uses rebasing, because it leads to a cleaner commit history.

First you need to get the latest upstream changes to your local copy. This can be achieved with the `upstream` setup listed earlier.

```
$ git fetch upstream
```

Then we tell Git to move our commit on top of the new master repository.

```
$ git rebase upstream/master
```

Git will try to apply your commits, fail and exit with an error message listing which files have conflicts. You need fix all the problems either by manual editing or with a dedicated diff tool. After you have fixed all the issues, run the full test suite to ensure you have not broken anything by accident.

Rebasing is finished by adding all files that had conflicts:

```
$ git add file1.py file2.py
```

and then telling Git to finish the operation:

```
$ git rebase --continue
```

The updated result can now be pushed to Github just like in the previous chapter:

```
$ git push --force
```

The pull request is updated and CI run automatically.

A.5 Documentation updates

All of Meson's documentation, including the entire <https://mesonbuild.com> web site is stored in the same repository as the source code. Commits that add new functionality (as opposed to just fixing bugs) must update all documentation at the same time. Just like with missing unit tests, missing documentation is reason enough to reject any contribution.

There are three separate classes of documentation. These are the user manual, the reference documentation and release notes. The manual is written in plain English and explains the concepts and basic usage of each feature. The reference documentation on the other hand explains features in detail, but is written in a terse style. Release notes are short pieces of text describing new features added in Meson since the last major release.

All documentation can be found in the `docs` subdirectory. They are written in Markdown format and are processed to create the final documentation using the HotDoc [6] document generation program. Thus the reader is assumed to be familiar with Markdown syntax. There are many good tutorials available on the Internet.

Reference documentation changes are the most common. These include adding new functions, keyword arguments to existing functions and so on. The actual change is done by finding the suitable location in the reference documentation page and adding your changes. Changing the manual needs to be done less often, but it does take more work. Usually you need to write some text explaining the new feature and an example of how you would use it in a project.

Changes in these documentation pages get updated on the web site fairly quickly, sometimes on the same day as they are merged. Thus the documentation may refer to features that are not yet in any released version. Because of this all new features must declare which versions of Meson they are available in. This is done by a piece of text saying *This feature is available since version X*, where *X* is the next major release of Meson.

Release notes follow a slightly different pattern. They are stored in standalone files that do not show up on the web site. When a release is published, the web site is updated with a release note page for the new release that contains all the individual files. These are called *snippets* and they are stored in `docs/markdown/snippets` subdirectory. The files must conform to the following format:

Headline describing the feature

A paragraph or few of text describing the new feature and how it can and should be used. Sample code should be written in this special format:

```
```meson
variable = new_awesome_function()
```
```


Bibliography

- [1] Bob Amstadt, Eric Youngdale, and Alexandre Julliard. Wine. <https://www.winehq.org/>.
- [2] Fabrice Bellard. The QEMU emulator. <https://www.qemu.org/>.
- [3] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorff. The CMake build system. <https://cmake.org/>, 2000.
- [4] John Donne. *Devotions Upon Emergent Occasions*. 1624.
- [5] Ulrich Drepper. How to write shared libraries. <https://akkadia.org/drepper/dsohowto.pdf>, December 2011.
- [6] Mathieu Duponchelle. The HotDoc API documentation system. <https://hotdoc.github.io/>.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1994.
- [8] James Henstridge. Pkg-config. <https://www.freedesktop.org/wiki/Software/pkg-config/>, 2000.
- [9] Charles Antony Richard Hoare. Unification of theories: A challenge for computing science. In *Selected papers from the 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop on Recent Trends in Data Type Specification*, pages 49–57. Springer–Verlag, 1996.
- [10] Michael Kerrisk. *The Linux programming interface*. No Starch Press, 2010.
- [11] Helmut Kopka and Patrick W. Daly. *A Guide to L^AT_EX 2_ε, Document Preparation for Beginners and Advanced Users*. Addison–Wesley Publishers Ltd., second edition, 1995.

- [12] Alexander Larsson. The Flatpak packaging system. <http://flatpak.org/>, 2015.
- [13] Evan Martin. The Ninja build system. <https://ninja-build.org/>, 2012.
- [14] Microsoft. Visual Studio. <https://visualstudio.microsoft.com/>.
- [15] Haavard Nord and Eirik Chambe-Eng. The Qt development framework. <https://www.qt.io/>, 1995.
- [16] Julian Seward. The Valgrind program analysis framework. <http://valgrind.org/>.
- [17] Richard Stallman. The GNU debugger. <https://www.gnu.org/software/gdb/>, 1986.
- [18] Tool Interface Standard. Executable and linking format (ELF) specification version 1.2, 1995.
- [19] Wikipedia contributors. Turing completeness — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Turing_completeness, 2018.

Index

- <3, 38
- ?:, *see* ternary operator
- \$@, 141

- abstract machine, 23
- accessing resources, 108
- add_global_arguments(), 229
- add_global_link_arguments(), 230
- add_languages(), 231
- add_project_arguments(), 232
- add_project_link_arguments(), 232
- add_test_setup(), 232
- address layout randomisation, 20
- alias_target(), 233
- API design, 160
- Apple frameworks, 60
- application bundle, 101
- Arduino, 133
- arguments
 - keyword, 35, 36
 - positional, 35
- arithmetic, 28
- array object, 183
 - contains(), 184
 - get(), 185
 - length(), 185
- ASLR, *see* address layout randomisation
- assert(), 234
- asset pipeline, 49, 119

- bare metal, 133, 137

- BAT files, 26
- benchmark(), 234
- benchmarks, 97
- boolean object, 29, 185
 - to_int(), 185
 - to_string(), 186
- Boost, 58
- both_libraries(), 235
- build machine, 132
- build options, 111
- build phases, 31
- build_machine object, 195
 - cpu(), 196
 - cpu_family(), 195
 - endian(), 197
 - system(), 196
- build_target(), 235
- build_target object, 197
 - extract_all_o...(), 197
 - extract_objects(), 197
 - full_path(), 198
 - name(), 198
 - private_dir_i...(), 198

- Canadian Cross, 132
- cargo cult, 18
- circular dependencies, 18, 26, 45
- CMake, 59, 269
- cmake module, 269
 - configure_pack...(), 270
 - write_basic_p...(), 269
- COBOL, 25

- command argument strings, 122
- compile time, 9
- compiler flags, 73
- 300 **compiler object**, 198
 - `alignment()`, 198
 - `check_header()`, 199
 - `command_array()`, 199
 - `compiles()`, 199
 - `compute_int()`, 200
 - `find_library()`, 200
 - `first_supporte...()`, 202
 - `first_supporte...()`, 202
 - `get_id()`, 201
 - `get_linker_id()`, 202
 - `get_supported...()`, 202
 - `has_argument()`, 202
 - `has_function()`, 203
 - `has_function...()`, 203
 - `has_header()`, 203
 - `has_header_sy...()`, 203
 - `has_link_argu...()`, 204
 - `has_member()`, 204
 - `has_members()`, 205
 - `has_multi_arg...()`, 205
 - `has_multi_lin...()`, 205
 - `has_type()`, 205
 - `links()`, 205
 - `run()`, 206
 - `sizeof()`, 206
 - `symbols_have...()`, 206
 - `version()`, 207
- configurability, 71
- configuration data object**, 207
 - `get()`, 207
 - `get_unquoted()`, 208
 - `has()`, 208
 - `merge_from()`, 209
 - `set()`, 209
 - `set10()`, 210
 - `set_quoted()`, 210
- configuration header, 74
- `configuration_data()`, 75, 238
- `configure_file()`, 75, 238
- continuous integration, 87, 294
- CPU, 137
- CPU family, 137
- cross compilation, 131
 - of a cross compiler, 134
- cross file, 135, 179
- Cups, 60
- `custom_command()`, 119
- `custom_target()`, 240
- custom_target object**, 211
 - `full_path()`, 211
 - `to_list()`, 211
- Cygwin, 196
- D-Bus, 276
- `declare_dependency()`, 66, 242
- dependency**, 55
 - backend, 58
 - circular, *see* circular dependencies
 - executable, 60
 - overriding, 67
 - fallback, 67
 - internal, 66
 - manual lookup, 61
 - modules, 57
 - native, 142
 - vendoring, 145
 - via a Wrap, 146
- `dependency()`, 56, 58, 244
- dependency file, 121, 239
- dependency object**, 211
 - `as_system()`, 211
 - `found()`, 212
 - `get_pkgconfig...()`, 212
 - `get_variable()`, 212
 - `is_system()`, 212
 - `name()`, 212
 - `partial_depend...()`, 213
 - `type_name()`, 213
 - `version()`, 213

- DESTDIR, 105
- dictionary object, 30, 186
 - get(), 187
 - has_key(), 186
- disabler, 42
- disabler(), 245
- disabler object, 187
 - found(), 188
- dlang module, 271
 - generate_dub_...(), 271
- DLL, 102
- Doxygen, 56
- Ducky library, 159
- edit-compile-test cycle, 31
- Einstein, Albert, 25
- ELF, 21, 108
- ELF interpreter, 17
- endianness, 137
- environment(), 246
- environment object, 214
 - append(), 214
 - prepend(), 215
 - set(), 215
- environment variables, 5, 60, 89, 106, 214
- error(), 247
- event driven IO, 80
- exe wrapper, 95, 140, 233
- executable(), 36, 246
- external_library object, 215
- external_program object, 215
 - found(), 215
 - path(), 215
- files(), 248
- find_program(), 60, 68, 247
- firmware, 55
 - flashing, 143
- Flatpak, 101
- foreach, 40, 82
- fs (filesystem) module, 271
 - exists(), 272
 - hash(), 272
 - is_dir(), 272
 - is_file(), 272
 - is_samepath(), 273
 - is_symlink(), 273
 - name(), 273
 - parent(), 273
 - replace_suffix(), 274
 - size(), 274
- functional programming, 26
- functions, 35
- generator(), 127, 249
- generator object, 215
 - process(), 216
- generators, 127
 - extra arguments, 129
 - file layout, 120
 - when cross compiling, 141
- get_option(), 250
- get_variable(), 250
- Git, 146, 291
 - force pushing, 294
 - Github, 291
 - rebasing, 295
- global offset table, 21
- gnome module, 274
 - compile_resources(), 274
 - compile_schemas(), 276
 - gdbus_codegen(), 276
 - generate_gir(), 277
 - generate_vapi(), 278
 - genmarshal(), 278
 - gtkdoc(), 279
 - gtkdoc_html_dir(), 280
 - mkenums(), 280
 - mkenums_simple(), 280
 - yelp(), 281
- GNU Autotools, 154
- GNU debugger, 96
- GNU triplet, 134

- Golgafrincham proverbs, 169
- GPG, 171
- Gtk-Doc, 56
- halting problem, 32
- header platform detection, 72
- header search directories, 51
- Hello, world, 6
- host machine, 132
- `host_machine` object, 216
- hotdoc module, 281
 - `generate_doc()`, 282
- i18n module, 282
 - `gettext()`, 282
 - `merge_file()`, 283
- IDL, 48
- if clause, 38
- immutability, 25, 32, 41
- `import()`, 251, 269
- `include_directories()`, 251
- incremental builds, 32
- `install_data()`, 103, 252
- `install_headers()`, 253
- `install_man()`, 254
- `install_subdir()`, 103, 255
- integer object, 28, 188
 - `is_even()`, 188
 - `is_odd()`, 188
- IRC, 291
- `is_disabler()`, 256
- `is_variable()`, 257
- `jar()`, 257
- Java, 257
- `join_paths()`, 258
- keyword arguments, *see* arguments, key-word
- kwargs, *see* arguments, keyword
- language runtime library, 12, 17
- `LD_LIBRARY_PATH`, 16
- `library()`, 258
- linker, 10
 - dynamic, 11, 16
 - GNU bfd, 19
 - ld, 11
 - path lookup, 15
 - shared, 15
 - static, 10, 14
- linker script, 237
- LLVM, 19
- loader, 21
- Lua, 246
- LZMA, 169
- Make, 24, 154, 155
- make
 - recursive, 152
- Man pages, 103, 254
- man pages, 174
- Markdown, 296
- math library, 62, 201
- merge conflicts, 294
- merge requests, 293
- Meson language, 23
- `meson` object, 217
 - `add_dist_script()`, 217
 - `add_install_s...`, 217
 - `add_postconf...`, 217
 - `backend()`, 217
 - `build_root()`, 218
 - `can_run_host...`, 218
 - `current_build...`, 218
 - `current_source...`, 218
 - `get_compiler()`, 218
 - `get_cross_pro...`, 219
 - `get_external...`, 220
 - `has_exe_wrapper()`, 220
 - `install_depend...`, 220
 - `is_cross_build()`, 220
 - `is_unity()`, 220
 - `override_depen...`, 221
 - `override_find...`, 221

- `project_licenses()`, 222
 - `project_name()`, 222
 - `project_version()`, 222
 - `source_root()`, 222
 - `version()`, 222
- `message()`, 35, 258
- MinGW, 131
- monorepo, 145
- MSI packages, 5
- native files, 139
- native target, 142
- Ninja, 7
- nondeterminism, 126
- not a number, 42
- null type, 31
- object
 - domain specific, 195
- objects, 41
 - elementary, 183
- opaque types, 161
- OpenGL, 60
- `option()`, 112, 229
- option types, 112–114
- options
 - base, 116
 - core, 115
 - directory, 116
 - feature, 113
 - language, 116
 - listing, 114
 - project, 116
 - setting, 114
 - yielding, 117
- package manager, 3, 100
- path joining, 27, 189
- Pkg-config, 58, 213
 - generating, 169
- `pkgconfig` module, 284
 - `generate()`, 284
- platform bootstrapping, 131
- poetry, 55
- position independent code, 16
- positional arguments, *see* arguments,
 - positional
- precompiled headers, 163, 236
- preprocessor, 53
- procedure linkage table, 21
- project releases, 169
- PyPi, 178, *see* Python, Package Index
- Python, 4, 26, 61
 - extension modules, 177, 285
 - Package Index, 178
 - Pip, 4
- python module, 285
 - `find_installation()`, 286
- python_installation object, 222
 - `dependency()`, 223
 - `extension_module()`, 223
 - `get_install_dir()`, 223
 - `get_path()`, 223
 - `get_variable()`, 224
 - `has_path()`, 224
 - `has_variable()`, 224
 - `install_sources()`, 223
 - `language_version()`, 223
- QEMU, 140
- QMake, 59
- Qt, 57
- qt4 module, 287
- qt5 module, 286
 - `compile_transl...()`, 287
 - `preprocess()`, 286
- quoting, 26, 208
- race condition, 126
- Raspberry Pi, 133
- `reinterpret_cast`, 164
- relocation record, 20
- resource bundling, 123
 - Gnome, 274

- Qt, 287
- Windows, 288
- return code, 90, 92, 224
- rpath, 108
- rpm module, 287
- run time, 9
- run_command(), 259
- run_result object, 224
 - compiled(), 224
 - returncode(), 224
 - stderr(), 224
 - stdout(), 225
- run_target(), 259
- SDL, 60
- set_variable(), 260
- shared library versioning, 261
- shared_library(), 260
- shared_module(), 261
- SIMD, 90
- soname, 21
- source code generation, 48, 68, 123
- source_configuration object, 225
 - dependencies(), 225
 - sources(), 225
- source_set object, 225
 - add(), 225
 - add_all(), 226
 - all_dependencies(), 226
 - all_sources(), 226
 - apply(), 226
- sourceset module, 288
 - source_set(), 288
- stable ABI, 159
- static linker, *see* linker, static
- static_library(), 262
- string object, 27, 189
 - contains(), 190
 - endswith(), 190
 - format(), 28, 190
 - join(), 191
 - split(), 191
 - startswith(), 192
 - strip(), 192
 - to_int(), 192
 - to_lower(), 193
 - to_upper(), 193
 - underscorify(), 193
 - version_compare(), 194
- strings
 - concatenating, 27
 - file names as, 189
- subdir(), 33, 262
- subdir_done(), 263
- subdirectories, 33, 50
- subproject(), 64, 263
- subproject object, 228
 - found(), 228
 - get_variable(), 65, 228
- subprojects, 63
 - obtaining via wraps, 146
- summary(), 264
- symbol name mangling, 207
- symbol resolution, 12, 16
- symbol visibility, 161, 236
- system introspection, 61, 78, 116
- target machine, 132
- target properties, 52
- target_machine object, 228
- ternary operator, 39
- test
 - setup, 96
- test(), 88, 265
- tests
 - arguments, 88
 - environment variables, 89
 - exe wrapper, 95
 - failing, 92
 - parallelism, 91
 - skipping, 87, 90
 - suite, 86, 94
 - timeout, 87, 90, 95
 - tool, 93

- texture atlas, 121
- threads, 62
- translation targets, 283
- Turing completeness, 25, 41
- undecidability, 32
- undefined behaviour, 64, 165
- Unix file system hierarchy, 99
- Unix shell, 26, 45
- Vala, 278
- Valgrind, 95, 233
 - suppressions, 96
- variables, 24
 - scope, 51
- `vcs_tag()`, 267
- visibility attributes, 162
- Visual Studio, 7, 161
- `windows` module, 288
 - `compile_resources()`, 288
- Wine, 140
- Wrap, 146
 - command line tool, 148
- WrapDB, 148
- `XDG_DATA_DIRS`, 137
- `XDG_DATA_HOME`, 137
- XML, 157
- Zlib, 56, 147, 169