Earthly Technologies

MAKEFILE
INS AND OUTS

Contents

Makefile Tutorials and Examples to Build From

Building software is a multi-step process—installing or updating dependencies, compiling the source code, testing, installing, and so on. In any moderately sized project, you might find it difficult to perform all these steps manually. This is where `make` can help you.

The `make` tool automates compilation of the software from the source code. It won't repeat a step if none of its prerequisites has changed, thus saving you time and resources.

In this chapter, you will learn how to write a simple Makefile and learn about important components of `make`, including variables, pattern rules, and virtual paths. You will also see some examples of using `make` with different technologies.

The Makefile

When you run `make`, it looks for a file named `Makefile`, or `makefile` in the same directory. The name `Makefile` is suggested so that it appears near other important files such as `README`.

You can name your Makefile anything, but then you have to explicitly tell `make` which file to read:

```
make -f some_other_makefile
```

The Makefile should consist of one or more rules. Each rule describes a goal or a step in your build process, the prerequisites for that step, and recipes for how to execute it.

The format for each rule is as follows:

```
target1 [target2 ...]: [pre-req1 pre-req2 pre-req3 ...]
    [recipes
    ...]
```

The parts in `[]` are optional. Each rule must have one or more targets, zero or more prerequisites, and zero or more recipes. The `target` is the file you want to be created in that rule. The prerequisites can be the name of an existing rule, or the name of a file in the same directory. The recipes are shell commands that need to be run in order to generate the target.

When `make` executes a rule, it looks at the prerequisites. If all the prerequisites are older than the target file, it means that none of them has changed since the last time the rule was executed. So `make` does not execute the rule. If, however, any prerequisite is newer than the target, the recipes are executed.

Here's an example. Create a file named `data.txt` with the text `hello world.` You'll use the `wc` command to calculate the number of characters, words, and lines and store it in a file named `count.txt`. In this simple demonstration, you have a dependency and a target that needs to be built from the dependency.

First, let's do it manually.

```
wc -c data.txt > count.txt # Count characters
wc -w data.txt >> count.txt # Count words
wc -l data.txt >> count.txt # Count lines
```

This should create a file named `count.txt` with the following content:

```
13 data.txt
2 data.txt
0 data.txt
```

Let's write the Makefile to automate this:

```
all: count.txt

count.txt: data.txt
    wc -c data.txt > count.txt # Count characters
    wc -w data.txt >> count.txt # Count words
    wc -l data.txt >> count.txt # Count lines
```

This Makefile has two targets. The first target is `all`, which acts like an overall build target. It is not necessary to have such a target, especially when our build has only one step, but it is a recommended practice.

The `all` target depends on `count.txt` and has no recipe. This means that `all` will be prepared as soon as `count.txt` is prepared.

The target `count.txt` depends on the file `data.txt` and the recipes list contains the commands you ran previously.

Now, run `make` again from the terminal. You should see that `make` executes the commands listed and creates `count.txt`. If you run the `make` command again, you should see the output:

```
make: Nothing to be done for 'all'.
```

Let's break it down. When you run `make` without any argument, it runs the first target, which is `all` in this case. Since `all` depends on `count.txt`, that target is executed. The target `count.txt` depends on `data.txt`, so the commands are run and the file is generated.

The next time you run `make` following the same sequence, `make` looks at `count.txt` and notices that `count.txt` is newer than `data.txt`, meaning the dependency has not been changed since the last time `make` was run, so it doesn't do anything.

Edit the `data.txt` file and change the text to `hi world`. Now when you run `make`, it runs the commands and updates `count.txt`. Since the dependency was changed, it rebuilt the target.

You can also run a target directly by passing its name to the `make` command. Running `make count.txt` will run only the `count.txt` rule.

Let's add a rule to clean the project files. It is a recommended practice to have a `clean` rule to delete any generated files, effectively returning the project to the initial state. Add the following rule to your Makefile:

```
clean:
    rm count.txt
```

The `clean` rule doesn't have a prerequisite. The targets without a prerequisite are considered to be older than their dependencies, and so they're always run.

Components of Makefile

Here are some important components that can help you write more concise and simpler Makefiles.

Comments

You can have comments in Makefile that start with a `#` and last till the end of the line.

```
all: count.txt # This is a comment
...
```

Variables

Just like regular programming languages, `make` supports using variables to avoid repetitions and keep the Makefile clean. Another advantage of variables is that the user can override them without needing to edit the Makefile manually.

A variable in Makefile starts with a $ and is enclosed in parentheses () or braces {}, unless it's a single character variable. To set a variable, write a line starting with a variable name followed by `=`, `:=` or `::=`, followed by the value of the variable:

```
TARGET = count.txt
SOURCE = data.txt
```

The variables defined with `=` are called "recursively expanded variables," and those defined with `:=` and `::=` are called "simply expanded variables." There is a subtle difference between these two, which you can read about in the manual.

You can reference these values in any of the targets, prerequisites, or recipes:

```
TARGET = count.txt
SOURCE = data.txt

all: $(TARGET)

$(TARGET): $(SOURCE)
    wc -c $(SOURCE) >  $(TARGET) # Count characters
    wc -w $(SOURCE) >> $(TARGET) # Count words
    wc -l $(SOURCE) >> $(TARGET) # Count lines

clean:
   rm $(TARGET)
```

Here instead of hard-coding the target and source file names, we have used two variables, with default values of `count.txt` and `data.txt`. If you run the `make` command, it should work just like before. However, if you want to change the name of the target to, for example, `newcount.txt`, you can do so without changing the Makefile:

```
make TARGET=newcount.txt
```

Passing `TARGET=newcount.txt` overrides the default value of `$(TARGET)` in the Makefile and so, instead of `count.txt`, the file `newcount.txt` is generated. Similarly, you can run `make`

TARGET=newcount.txt clean to clean this new file.

When make is run, it also converts all available environment variables into make variables. So you can freely use any environment variable.

Automatic Variables There are some special variables called automatic variables. Their values are computed each time for every rule and are based on the target and prerequisite file names. Here are some of the most important automatic variables:

1. **$@**: This is the target file name. If there is more than one target, this is whichever target caused the recipe to run.
2. **$***: This is the target file name without the extension.
3. **$<**: This is the name of the first prerequisite.
4. **$?**: The names of all the prerequisites that are newer than the target, with spaces between them. If the target does not exist, all prerequisites will be included.
5. **$^**: The names of all the prerequisites, with spaces between them and duplicates removed.
6. **$+**: Same as $^, except it includes duplicates.

There are other automatic variables. For a full list, see the manual.

Using the automatic variables, we can simplify our Makefile a bit more:

```
TARGET = count.txt
SOURCE = data.txt

all: $(TARGET)

$(TARGET): $(SOURCE)
	wc -c $< >  $@ # $< matches the source file name, $@ matches the target file name
	wc -w $< >> $@
	wc -l $< >> $@

clean:
	rm $(TARGET)
```

Virtual Paths

Often you have files organized into directories. It is not always possible to write the entire file name every time. You can use VPATH to specify where make should search for targets and prerequisites.

For example:

```
VPATH = src include

foo.o: foo.cpp
```

Here make will search for foo.o and foo.cpp first in the current directory, and if not found will look inside the directories listed in VPATH. Thus if you have src/foo.cpp, instead of writing the whole path every time, you can use VPATH to tell make where to search for it.

However, there is a slight issue. Usually the `cpp` files are stored under `src`, while the header files are stored under `include`. But in our previous example, `make` searches for `foo.cpp` in both of those directories. You can tell `make` that `cpp` files should be searched in `src` and headers should be searched in `include`. For that, `vpath` (note: lowercase) is used:

```
vpath %.cpp src
vpath %.h include
```

The `%` is like `*` of regex. It matches anything. The previous rule tells `make` to search for files ending in `.cpp` in `src` and files ending in `.h` in `include`.

Pattern Rules

A pattern rule contains the character `%` exactly once. The `%` matches any character. For example, `%.cpp` matches any files ending in `.cpp`, while `a%b` matches any file starting in `a` and ending in `b` and having anything in between, like `axb` or `axyzb`, but not `ab`. There should be at least one character to match `%`. The part that matches the `%` is called the stem.

When used in a prerequisite, the `%` stands for the same stem that was matched by the `%` in the target. For example:

```
%.o: %.cpp
    ...
```

This tells how to make `x.o` from `x.cpp` where `x` stands for anything, provided `x.cpp` should exist or can be made. So if you have `a.cpp` and `b.cpp`, that single rule can make both `a.o` and `b.o`.

Phony Target

In our Makefile, there are two “special” targets—`all` and `clean`. Since they do not have any prerequisite, and there are no files named `all` or `clean` in the project, they are always considered to be older than their dependencies and always executed.

But if you create a file called `all` or `clean` in the directory, `make` will get confused. Since the `all` or `clean` file is there, and the targets have no prerequisites, they will be considered newer than their prerequisites. Therefore, the recipes will never be run. To fix this, you can declare the targets to be “phony”:

```
.PHONY: all clean

...
```

For the full manual of `make`, read the `make` documentation.

Next Up

The `make` tool is a valuable one to master in software development. Using it can speed up your development and ensure an easier process overall. However, due to its feature-rich nature, `make` can be hard to master.

In the following chapters we will get you up to speed on make and makefiles.

Creating a G++ Makefile

C++ is one of the most dominant programming languages. Although there are many compilers available, GCC still ranks as one of the most popular choices for C++. GCC is part of the GNU toolchain, which comes with utilities like GNU make, GNU bison, and GNU AutoTools.

What Is GCC?

GNU Compiler Collection, also known as GCC, started as a C compiler, created by Richard Stallman in 1984 as a part of his GNU project. GCC now supports many languages, including C++, Objective C, Java, Fortran, and Go. The latest version as of writing this chapter is GCC 11.1, released April 27, 2021.

The C++ compiler of GCC is known as `g++`. The `g++` utility supports almost all mainstream C++ standards, including `c++98`, `c++03`, `c++11`, `c++14`, `c++17`, and experimentally `c++20` and `c++23`. It also provides some GNU extensions to the standard to enable more useful features. You can check out the detailed standard support on gnu.org.

In this tutorial, you will learn how to compile C++ programs with the `g++` compiler provided by GCC, and how to use Make to automate the compilation process.

Installing GCC

I'll touch briefly on installing for Linux, Mac, and Windows.

Linux

GCC is one of the most common tools in the unix world, and is available in every single Linux distribution. Here, I show you how to install the GNU toolchain for some famous distributions.

For Ubuntu, you need to run the following command:

```
sudo apt update && sudo apt install build-essentials
```

For Arch Linux, run:

```
sudo pacman -S base-devel
```

For Fedora, run:

```
dnf groupinstall 'Development Tools'
```

For other distributions, consult the official wiki of your distribution.

Mac

To install GCC on Mac, run `brew install gcc` which will place `g++-11` in `/usr/local/bin`. Then create an alias to `g++`: `alias g++='g++-11'`.

Windows

To use GCC in Windows, use WSL2. You can install GCC inside the Windows Subsystem for Linux (WSL) and use it from there.

Compiling With G++

Let's take a look at how the compilation with G++ works. You will compile a simple `Hello, World!` program. Save the following file as `hello.cpp`:

```cpp
#include<iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

To compile this file, simply pass this file to `g++`:

```
g++ hello.cpp
```

By default, `g++` will create an executable file named `a.out`. You can change the output file name by passing the name to the `-o` flag.

```
g++ -o hello hello.cpp
```

This will compile `hello.cpp` to an executable named `hello`. You can run the executable and see the output:

```
./hello
```

The Compilation Process

Although the compilation can be done with one command, the compilation process can be divided into four distinct phases:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

In the preprocessing part, the GNU preprocessor (`cpp`) is invoked, which copies the header files included via `#include`, and expands all macros defined with `#define`. You can perform this step manually by running the `cpp` command.

```
cpp hello.cpp > hello.i
```

The file `hello.i` contains the preprocessed source code.

In the next phase, the `g++` compiler compiles the preprocessed source code to assembly language. You can run this step manually with the following command:

```
g++ -S hello.i
```

The `-S` flag creates a file `hello.s`, which contains the assembly code.

In the next step, the assembler `as` converts the assembly to machine code.

```
as -o hello.o hello.s
```

Finally, the linker `ld` links the object code with the library code to produce an executable.

```
ld -o hello hello.o ...libraries...
```

The `libraries` argument above is a long list of libraries that you need to find out. I omitted the exact arguments because the list is really long and complicated, and depends on which libraries `g++` is using on your system. If you are interested to find out, you can run the command `g++ -Q -v -o hello hello.cpp` and take a look at the last line where `g++` invokes `collect2`

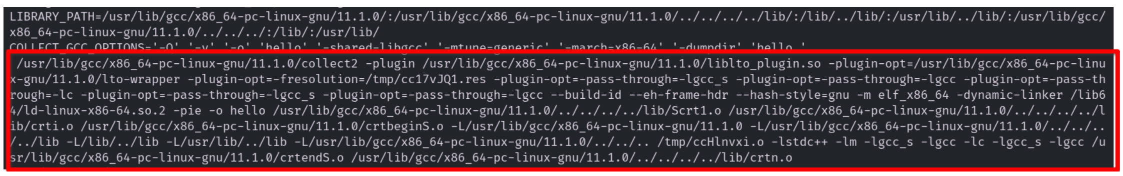


Figure 1: Libraries Used By g++

Thankfully, you do not have to perform these steps manually, as invoking `g++` itself will take care of all these steps.

Using the `make` Utility

Even though the compilation commands have been simple so far, this is not necessarily the case when you have multiple source files. As an example, consider this program:

```cpp
#include "func.h"

int main() {
    func(10);
    func(100);

    return 0;
}
```

`hello.cpp`

This file includes `func.h`, which contains the declaration for a simple function:

```cpp
#ifndef MAKE_GPP_FUNC
#define MAKE_GPP_FUNC

#include<iostream>

void func(int i);

#endif
```

`func.h`

Finally, the definition of `func` resides in `func.cpp`:

```
#include "func.h"

void func(int i) {
    std::cout << "You passed: " << i << std::endl;
}
```

func.c

In order to compile your program, you need to compile both `hello.cpp` and `func.cpp`, since the former depends on the latter.

```
g++ -o hello hello.cpp func.cpp
```

If you have more files, then you need to list all of them, while taking care to set the correct include paths and library paths. Moreover, if your code uses any library, you need to list those libraries, too. The resultant command is likely massive and difficult to remember and type. Also, the compilation command will compile all of the source files every time it is executed. But if some of the source files haven't been modified since the last compilation, it's a waste of time and resources to compile all the files. But keeping track of what has changed manually is also a difficult task.

This is where the `make` utility helps. `make` lets you define your target, and how to reach the target and what are the dependencies. Then it automatically keeps track of which dependencies have changed and recompiles only the necessary parts.

So let's see how you can utilize `make`.

The Makefile

In order to let `make` know what to do, you need to create a file named `Makefile` in the root of your project. This file can also be named `makefile` but is traditionally named `Makefile` so that it appears near other important files such as `README`.

Create an empty `Makefile` in the project root and run the command `make` from the project directory. You should see the following output:

```
make: *** No targets. Stop.
```

It means `make` has found the Makefile, but since it is empty, it doesn't know what to do.

Now let's see how you can utilize Makefile to tell `make` what to do. The Makefile consists of a set of rules. Each rule has three parts—a target, a list of prerequisites, and a recipe—like this:

```
target: pre-req1 pre-req2 pre-req3 ...
    recipes
    ...
```

Note that there are tabs before the recipe lists. You can't use any other whitespace character. You must use tabs.

When `make` executes a target, it looks at its prerequisites. If those prerequisites have their own recipes, `make` executes them and when all the prerequisites are ready for a target, it executes the corresponding recipe for the current target. For each target, the recipes are executed only if the target doesn't exist or the prerequisites are newer than the target.

Let's update the `Makefile` for the example program:

```
all: hello

hello: hello.o func.o
 g++ -o hello hello.o func.o

func.o: func.cpp func.h
 g++ -c func.cpp

hello.o: hello.cpp
 g++ -c hello.cpp
```

Now, run the `make` command again. You should see the commands being run by `make`:

```
g++ -c hello.cpp
g++ -c func.cpp
g++ -o hello hello.o func.o
```

And you'll notice that an executable called `hello` has been created in the directory. So, how did `make` do that? Let's analyze.

When you run `make` without any arguments, it executes the first target. In the Makefile, the `all` target has a prerequisite `hello`. So, `make` looks for a rule to create `hello`. The rule `hello` has two prerequisites `hello.o` and `func.o`. Now, the target `hello.o` depends on `hello.cpp` which exists and is newer than the target `hello.o` (which does not exist). So, `make` now executes the recipe for `hello.o` and runs the command `g++ -c hello.cpp`. This creates the `hello.o` file.

Now `make` starts resolving `func.o`. Both of its pre-requisites exist and are newer than the target. So `make` executes the command `g++ -c func.cpp`. Now that the target `hello` has both the prerequisites satisfied, its recipe can be executed and the `hello` file is created.

Now what happens if one of the files is changed? Let's change the `hello.cpp` file and change the `func(10)` line to `func(20)`:

```
#include "func.h"

int main() {
    func(20);
    func(100);

    return 0;
}
```

Now if you run `make`, you'll notice that it does not execute all the steps:

```
g++ -c hello.cpp
g++ -o hello hello.o func.o
```

This time, `make` does not compile `func.c` because the file `func.o` exists, and its prerequisites are not newer than itself. This is because you have not changed `func.cpp` or `func.h`.

On the other hand, the file `hello.cpp` is newer than `hello.o`. So it needs to be recompiled, and when `hello.o` is re-created, the target `hello` needs to be executed, since it depends on `hello.o`.

You can also call `make` with the name of a specific rule. For example, running `make func.o` will only run the rule for `func.o`

Comments in Makefile

You can have comments in Makefile, which start with a # and last till the end of the line.

```
all: hello # This is a comment

hello: hello.o
...
```

Using Variables

Observe that in your Makefile, there are quite a lot of repetitions. For example:

```
func.o: func.cpp func.h
 g++ -c func.cpp
```

In this rule, we have the string `func` repeated four times. Since here the base name of the source file and the compiled file are the same (`func`), we can use variables to tidy up the rules. The variables not only make the Makefile cleaner, they can be overridden by the user so that they can customize the Makefile without editing it.

A variable in Makefile starts with a `$` and is enclosed in parentheses `()`or braces `{}`, unless it's a single character variable.

To set a variable, write a line starting with a variable name followed by `=`, `:=` or `::=`, followed by the value of the variable:

```
objects = hello.o func.o
```

Here the variable `objects` is set to `hello.o func.o`. Now whenever you use this variable in a rule, it will be replaced by its value.

```
objects = hello.o func.o

all: hello

hello: $(objects)
    g++ -o hello $(objects)
```

This is the same as writing:

```
hello: hello.o func.o
    g++ -o hello hello.o func.o
```

There is another way of defining variables using the `?=` operator. This defines the variable only if it has not been defined before.

When you invoke `make`, it converts all the environment variables available to it with a `make` variable with the same name and value. This means you can set variables using environment variables. Also, you can override any variable by passing them while invoking make. For example, the `g++` command can be invoked through a variable.

```
CXX = g++
objects = hello.o func.o

all: hello

hello: $(objects)
    $(CXX) -o $(objects)
```

Now running `make` will compile the files with `g++`. However, the user can now substitute alternative if they want to.

```
make CXX=clang++
```

Now the files will be compiled by `clang++` since `CXX=clang++` overrides the variable `CXX` defined in the Makefile.

Phony Target

So far, you have only created files, but `make` can also "clean" files. Usually it's a good idea to have a `clean` target to delete all the generated files, basically returning the project to a clean slate.

Here is an example for your Makefile:

```
clean:
    rm *.o hello
```

You can run it via `make clean`. This cleans all the `.o` files and the `hello` file. Because the `rm` command does not create a file named `clean`, the `rm` command will be executed every time you invoke `make clean`.

But if you ever create a file called `clean` in the directory, `make` will get confused. Since the `clean` file is there, and the `clean` target has no prerequisites, it is always considered to be newer than its prerequisites. Therefore, the recipe will not run.

The same problem will arise with the `all` target if there is ever a file named `all`. To fix this, you can declare the targets to be "phony".

```
.PHONY: all clean

clean:
    rm *.o hello
```

Conclusion

Makefile is one of the most important components of compiling C++ using `g++`. It makes compilation easy and predictable and also saves time and resources by compiling only the necessary files. In this tutorial you learned how to install `g++`, and compile C++ programs with `g++`. You also learned how to write Makefiles and utilize `make` for increased productivity and automation.

Creating a Python Makefile

Even though Python is regarded as an interpreted language and the files need not be compiled separately, many developers are unaware that you can still use `make` to automate different parts of developing a Python project, like running tests, cleaning builds, and installing dependencies. It's honestly an underutilized function, and by integrating it into your routine, you can save time and avoid errors.

`make` is a commonplace tool in the world of software development, especially compiled languages like C or C++. It is a tool which controls the generation of executable and other non-source files from a program's source file. It can automate the process of building software by tracking its dependencies and compiling the program only when the dependencies change.

The reason `make` is very common with compiled languages is because the compilation commands for those languages can be long and complicated and difficult to remember. Also, you need to compile each file and link the resulting object files together. So, whenever one of the files changes, it becomes necessary to recompile it.

In this tutorial, you will learn the basics of `make` and how it can be used in a Python project.

Is Python Compiled or Interpreted?

Usually, all the programming languages can be classified into compiled or interpreted languages. In simple words, in a compiled language, the program is converted from a high level language to machine language. Whereas, in case of interpreted language, the source code is read and executed one line at a time.

Since in an interpreted language you cannot compile the source files independently beforehand, you cannot utilize `make` like you would do for a compiled language. This makes `make` relatively underutilized for an interpreted language.

Now, Python is usually considered to be an interpreted language. When you run a Python code, the Python interpreter reads the file line-by-line and runs it.

But behind-the-scenes, the source code is compiled into bytecode. These are similar to CPU instructions, but instead of being run by the actual CPU, these are executed by a software called a Virtual Machine (VM), which acts as a pseudo-microprocessor that runs the bytecodes. The advantage is that you can run Python on any platform as long as the VM is installed.

When you run a Python code, the interpreter implicitly compiles the code into bytecode and interprets it with the VM. The reason Python is regarded to be an interpreted language is because the compilation step is implicit. You don't have to invoke a compiler manually.

When you import a module into your code, Python compiles those modules into bytecode for caching purposes. These are stored in a directory named `__pycache__` in the current directory, which contains compiled `.pyc` files.

Although you cannot compile these modules using `make`, you can still use `make` for automation tasks like running tests, installing dependencies, cleaning the `.pyc` files etc.

Using Make With Python

In this tutorial, you'll create a simple app that makes requests to `http://numbersapi.com` and fetches random trivia about a user given number.

The Sample Source Code

First, you'll start by creating a file `api.py` with the following code:

```python
import requests

def get_fact(number):
    url = "http://numbersapi.com/{}".format(number)

    r = requests.get(url)
    if r.status_code == 200:
        print(r.text)
    else:
        print("An error occurred, code={}".format(r.status_code))
```

Create another file `app.py` with the following code:

```python
import api

api.get_fact(input("Enter a number: "))
```

This file simply imports the `api` module and calls the `get_fact()` function with a user provided number.

Finally, create a file `requirements.txt` with the dependencies of the app:

```
requests
```

This app only depends on the `requests` module, but a real life project possibly depends on a large number of modules. Let's run the app and see how it works.

First, install the dependencies:

```
pip install -r requirements.txt
```

Then, run the actual app:

```
python app.py
```

This will prompt you for a number and show you a trivia about that number:

```
$  python app.py
Enter a number: 20
20 is the number of questions in the popular party game Twenty Questions.
```

Now let's integrate `make` with our project to automate the installation of dependencies and running the app.

The Makefile

To use `make` in your project, you need to have a file named `Makefile` at the root of your project. This file instructs `make` on what to do. The Makefile consists of a set of rules. Each rule has 3 parts: a target, a list of prerequisites, and a recipe. They follow this format:

```
target: pre-req1 pre-req2 pre-req3 ...
	recipes
	...
```

Note that there are tabs before the recipe lists. Anything other than tabs will result in an error.

The `target` represents a goal that you want to achieve, usually this is a file that needs to be created in your build. The prerequisites list tells `make` which files are this target dependent on. The prerequisites can be a file or another target. Finally the recipes are a list of shell commands that will be executed by make as part of building the target.

When `make` executes a target, it looks at its prerequisites. If those prerequisites have their own recipes, `make` executes them and when all the prerequisites are ready for a target, it executes the corresponding recipe for the current target. For each target, the recipes are executed only if the target doesn't exist or the pre-requisites are newer than the target.

Our app has two targets. First, the dependencies must be installed and then the app can be run.

Let's create the rule for running the app first:

```
run:
	python app.py
```

The target is named `run` and it has no prerequisites, which it will be run every time you run `make run`. You can test it by running `make run` in a terminal.

Create another target for the setup stage:

```
setup: requirements.txt
	pip install -r requirements.txt
```

The `setup` target depends on the `requirements.txt` file. Whenever the `requirements.txt` file changes, the dependencies will be refreshed by running `pip install -r`.

Finally, let's have a `clean` rule to clean up the `__pycache__` folder:

```
clean:
	rm -rf __pycache__
```

Creating a Virtual Environment

The sample app depends on the `requests` library only. However, in a large project, there might be numerous dependencies. And if you are running multiple apps, it's possible that some apps require the same dependencies, but a different version. This means that one Python installation may not be capable of satisfying the requirements of all applications. The solution for this is to use a virtual environment. This is a self-contained directory tree that contains a Python installation of a specific version.

Different apps can use their own virtual environment where they can install their requirements. The virtual environments are isolated from each other, which means there will be no dependency conflicts.

Python provides a module called `venv` which is used to create and manage virtual environments. Let's see how a virtual environment can be used in the sample app.

First, you need to create a virtual environment in the project root:

```
python3 -m venv venv
```

This creates a `venv` folder in your current directory, which contains the necessary files to make the virtual environment.

There are two ways to use this virtual environment. Instead of using `python3` or `pip`, you have to use `./venv/bin/python3` or `./venv/bin/pip` to run the app or install dependencies:

```
python3 app.py # Uses the system Python
./venv/bin/python3 # USes the virtualenv Python
```

But writing `./venv.bin/` every time can be time consuming, especially if you have to run a lot of commands that use Python or pip. To overcome this, you can "activate" the virtual environment by running:

```
./venv/bin/activate
```

This will load the virtual environment in the current shell. This environment will stay active as long as you don't close the shell, or deactivate manually.

Once activated, running `python3` or `pip` will use the executables from the virtual environment. If you now run `pip install -r requirements.txt`, the modules will be installed in the `venv` directory.

Once you are done with the virtual environment, you can deactivate the environment by running the `deactivate` command.

`venv` in Make

You can utilize `make` to automatically refresh your virtual environment and run your app with this virtual environment. To automatically reinstall the dependencies whenever the `requirements.txt` file changes, write the following in your Makefile:

```
venv/bin/activate: requirements.txt
	python3 -m venv venv
	./venv/bin/pip install -r requirements.txt
```

Here the target is `venv/bin/activate` which depends on `requirements.txt`. Whenever `requirements.txt` changes, it rebuilds the environment and installs the dependencies with pip, which re-creates the `activate`. The actual goal here is the existence of the `venv` directory, but since `make` can only work with files, `venv/bin/activate` is used instead.

To run the app with this environment, create the following rule:

```
run: venv/bin/activate
 ./venv/bin/python3 app.py
```

The `run` target depends on the `venv/bin/activate` target. Once that target is satisfied, it runs the app using the virtual environment.

Finally, update the `clean` target to also delete the `venv` directory:

```
clean:
 rm -rf __pycache__
 rm -rf venv
```

Let's test this all out. First, delete the `venv` directory if you have one. Now run `make run`. Since the `venv/bin/activate` file does not exist, `make` will run the `venv/bin/activate` target, which will install the dependencies and finally run the app using the virtual environment.

```
→  make run
python3 -m venv venv
./venv/bin/pip install -r requirements.txt
Collecting requests
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Collecting idna<3,>=2.5
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting certifi>=2017.4.17
  Using cached certifi-2021.5.30-py2.py3-none-any.whl (145 kB)
Collecting urllib3<1.27,>=1.21.1
  Using cached urllib3-1.26.5-py2.py3-none-any.whl (138 kB)
Collecting chardet<5,>=3.0.2
  Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
Installing collected packages: idna, certifi, urllib3, chardet, requests
Successfully installed certifi-2021.5.30 chardet-4.0.0 idna-2.10 requests-2.25.1 urllib3-1.26.5
WARNING: You are using pip version 20.2.3; however, version 21.1.2 is available.
You should consider upgrading via the '/home/aniket/make-python/venv/bin/python3 -m pip install --upgrade pip' command.
./venv/bin/python3 app.py
Enter a number: 10
10 is the number of Provinces in Canada.
```

Figure 2: `make run` for Python `venv`

If you run `make run` once again, only the app will be run and the virtual environment will not be refreshed. You can use the `touch` command to stimulate a change in the `requirenenst.txt` file which will cause `make` to run the setup step again -

`touch requirements.txt`

Now if you run `make run`, the virtual environment will be recreated and then the app will be run.

Using Variables

Observe that in our Makefile, we have references to the `venv` directory in multiple places. In future, if we want to change the directory name to something else, we have to remember to perform the change in all the places. Also there isn't any way for the user to customize the directory name without editing the Makefile. To overcome this, we can use variables. The variables not only make the Makefile cleaner, they can be overridden by the user without editing Makefile.

A variable in Makefile starts with a $ and is enclosed in parentheses () or braces {}, unless its a single character variable.

To set a variable, write a line starting with a variable name followed by `=`, `:=` or `::=`, followed by the value of the variable:

```
VENV = venv
```

Here the variable `VENV` is set to `venv`. Now whenever you use this variable in a rule, it will be replaced by its value.

The rules now can be rewritten using the `VENV` variable:

```
VENV = venv
PYTHON = $(VENV)/bin/python3
PIP = $(VENV)/bin/pip

run: $(VENV)/bin/activate
 $(PYTHON) app.py


$(VENV)/bin/activate: requirements.txt
 python3 -m venv $(VENV)
 $(PIP) install -r requirements.txt


clean:
 rm -rf __pycache__
 rm -rf $(VENV)
```

I have also replaced references to `python3` and `pip` with a variable. By default, they will use the binaries from the virtual environment.

The variables can be overridden by providing the values when running the make command:

```
make VENV=my_venv run
```

Using `VENV=my_venv` overrides the default value of `VENV` and now the virtual environment will be created in `my_venv` directory. It is a good practice to use variables for all commands used in the Makefile, as well as their options and directories. This provides an easy way for the user to substitute alternatives.

Phony Targets

Your Makefile contains two special targets - run and clean. Special in the sense they don't represent an actual file that exists. And since `make` executes the recipes of a target if that target does not exist, these two targets will always be executed. However, if later if you have a file called `run` or `clean`, then since these targets have no prerequisites, `make` will consider these to always be newer and so, will not execute the recipes.

To overcome this, `make` has something called a Phony target. By declaring a target to be Phony, you tell `make` not to consider an existing file with the same name. To make the `run` and `clean` targets phony, add this to the top:

```
.PHONY: run clean
```

Conclusion

Using `make` in your Python projects opens the door to lots of possibilities in terms of automation. You can use `make` to run linters like `flake8`, run tests using `pytest`, or run code coverage using `coverage`. If you wish to learn all the features of `make`, be sure to check out the manual by GNU.

Creating a Golang Makefile

Building and testing any large codebase is time-consuming, error-prone, and repetitive. Golang supports multi-platform builds, which is excellent, but it needs multiple commands to build the binaries for different platforms, which means more time-consuming and repetitive steps when building binaries. If that's not enough, most projects have some dependencies that need to be installed before building the binary, and you probably want to run tests and ensure the code quality with linters and code coverage tools.

If this is starting to sound like a nightmare, rest assured: there is an easier way. The utility tool Make is used to automate tasks. It streamlines development and automates repetitive tasks with a single command. Make helps with testing, building, cleaning, and installing Go projects. In this tutorial, you will learn how you can leverage make and makefiles to automate all those frustrating and repetitive Golang tasks. You will learn how to build, clean, and test a Go sample project using make and a `Makefile`.

Adding a Makefile To Your Project

To start using make commands, you first need to create a `Makefile` in the root directory of your project. Let's create a simple `hello world` project with a `Makefile` in it.

`main.go`

```go
package main

import "fmt"

func main() {
 fmt.Println("hello world")
}
```

To run this project, you would normally need to build the project and run the binary:

```
go build main.go
```

If you want a different binary name and also want to create a build for a specific OS, you can specify this during the build:

```bash
GOARCH=amd64 GOOS=darwin go build -o hello-world main.go
```

You may want the build to create binary for multiple OS. For that, you will need to run multiple commands:

```
GOARCH=amd64 GOOS=darwin go build -o hello-world-darwin main.go
GOARCH=amd64 GOOS=linux go build -o hello-world-linux main.go
GOARCH=amd64 GOOS=windows go build -o hello-world-windows main.go
```

```
go run hello-world
```

The above commands can be simplified using Makefile. You can specify rules to a specific command and run a simple make command. You would not need to remember the commands and the flags or environment variables needed for executing it.

Makefile

```makefile
BINARY_NAME=hello-world

build:
	GOARCH=amd64 GOOS=darwin go build -o ${BINARY_NAME}-darwin main.go
	GOARCH=amd64 GOOS=linux go build -o ${BINARY_NAME}-linux main.go
	GOARCH=amd64 GOOS=windows go build -o ${BINARY_NAME}-windows main.go

run: build
	./${BINARY_NAME}

clean:
	go clean
	rm ${BINARY_NAME}-darwin
	rm ${BINARY_NAME}-linux
	rm ${BINARY_NAME}-windows
```

Now with these simple commands, you can build and run the Go project:

```
make run
```

Finally, you can run the clean command for the cleanup of binaries:

```
make clean
```

These commands are very handy and help to streamline the development process. Now all of your team members can use the same command. This reduces inconsistency and helps to eliminate project build-related errors that can arise with inconsistent manual commands.

Improving the Development Experience with Makefiles

`make` uses the Makefile as its source of commands to execute and these commands are defined as a *rules* in the Makefile. A single rule defines target, dependencies, and the recipe of the Makefile.

Terminology

- **Target:** Targets are the main component of a Makefile. The make command executes the recipe by its target name. As you saw in the last section, I used commands like `build`, `run`, and `build_and_clean`. These are called *targets*. Targets are the interface to the commands I want to execute.
- **Dependencies:** A target can have dependencies that need to be executed before running the target. For example, the `build_and_clean` command has two dependencies: `build` and `run`.
- **Recipe:** Recipes are the actual commands that will be executed when the target is run. A recipe can be a single command or a collection of commands. You can specify multiple commands in a target using a line break. In the example above, the recipe for the run target is `./${BINARY_NAME}`. A recipe should always contain a tab at the start.

Variables

Variables are essential to any kind of script you write. So Makefiles also have a mechanism to use variables. These are useful when you want the same configs or outputs to be used for different targets. In the example above, I have added the `BINARY_NAME` variable, which is reused across different targets.

The variable can be substituted by enclosing it `${<variable_name>}`. I have used the variable in the run command to execute the binary that was created from the build command:

```
BINARY_NAME=hello-world

run:
 ./${BINARY_NAME}
```

Variables can be defined either by using `=` or `:=`. `=` will recursively expand the variable. This will replace the value at the point when it is substituted. For example:

```
x = foo
y = $(x) bar
x = later

all:
 echo $(y)
```

When you run the `all` command, it will replace the value of `x` with the last updated value. The value has been changed to `later`, so it will print:

```
> later bar
```

The other kind of variable assignment is `:=`. These are *simple expanded* variables. The variable is expanded at the first scan. So if you assign the variable using this operator, it will print the first value:

```
x := foo
y := $(x) bar
x := later

all:
 echo $(y)
```

```
> foo bar
```

Some Useful Tips

- To make comments in a Makefile, you can simply add a `#` before any line.
- To disable printing the recipe while running the target command, use `@` before the recipe.
- Use the `SHELL` variable to define the shell to execute the recipe.
- Define the `.DEFAULT_GOAL` with the name of the target.

You can also define functions or loops in the Makefile. You can find more details on it in this make file tutorial.

Automating Tasks Using Makefile

While developing a project, you will have a lot of repetitive tasks that you might like to automate. In Golang, some of those tasks are testing, running test coverage, linting, and managing dependencies. I will be creating a Makefile that contains all the rules to automate these tasks:

```
BINARY_NAME=hello-world

build:
 GOARCH=amd64 GOOS=darwin go build -o ${BINARY_NAME}-darwin main.go
 GOARCH=amd64 GOOS=linux go build -o ${BINARY_NAME}-linux main.go
 GOARCH=amd64 GOOS=windows go build -o ${BINARY_NAME}-windows main.go

run: build
 ./${BINARY_NAME}

clean:
 go clean
 rm ${BINARY_NAME}-darwin
 rm ${BINARY_NAME}-linux
 rm ${BINARY_NAME}-windows

test:
 go test ./...

test_coverage:
 go test ./... -coverprofile=coverage.out

dep:
 go mod download

vet:
 go vet

lint:
 golangci-lint run --enable-all
```

With this simple Makefile, you can now easily execute commands to run tasks:

```
make test
make test_coverage
make dep
make vet
make lint
```

Note: I am using an external package, `golangci-lint`, for linting. If you are using `go mod`, make sure to add it to your `go.mod` file.

Any CI/CD tool that you are using can now simply use these targets.

Conclusion

Golang is a popular language for developing large-scale projects. Larger projects have multiple developers and require continuous automation to scale. Streamlining the development process by automating the tasks that are required during development, testing, and release will pay off with a faster and more reliable development process and a easier release process.

Building in Visual Studio Code with a Makefile

Microsoft announced recently a new Visual Studio Code extension to handle Makefiles. This extension provides a set of commands to the editor that will facilitate working with projects that rely on a Makefile to speed up the build.

In this tutorial, you'll set up a simple C++ project that depends on a well-known Python library to produce some sample charts. This is not a deep tutorial about make and Makefiles, but to get the most out of the extension you will need to have some concepts clear.

Make is one of the most used tools to build software projects, for good reason:

- You can get an implementation for almost any major operating system (POSIX/Windows/MacOS)
- It's technology agnostic. You can use it to build projects on any programming language (here's an example for JavaScript.
- Its task runner capabilities provide a multipurpose tool for almost any task.

A *Makefile* is a simple text file that defines rules to be executed. The usual purpose for Makefile in C++ projects is to recompile and link necessary files based on the modifications done to dependencies. However, Makefile and make are far more useful than that. The rules defined in a Makefile combine concepts like:

- Shell scripting
- Regular expressions
- Target notation
- Project structure

To illustrate this power, the sample project contains a single C++ source code file. The source code for the example is pretty simple —- it flips a coin as many times as the `iters` argument is passed, and then prints the number of heads and tails counted from each flip.

```
#include <cstdio>
#include <cstdlib>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

int flip_coins(int iters) {
  srand (time(NULL));
  int tails = 0;
  int heads = 0;

  for(int i=0;i < iters;i++){
      int coin = rand() % 2;
      if(coin == 1) {
          printf("heads\n");
          heads++;
      } else {
          printf("tails\n");
          tails++;
```

```
    }
  }
  printf("%d Heads, %d Tails\n",heads, tails);
  return abs(tails-heads);
}

int main(int argc,char* argv[]) {
    int iters =100;
    int diff = flip_coins(iters);
    if(100 < iters) {
        printf("With enough trials Heads should be close to Tails\n");
    }
}
```

This code will be compiled and linked with a simple Makefile that also will provide a couple of other standard rules for cleaning the compiled code and run a simple test.

Creating C++ Projects with VS Code

The VS Code extension **Makefile Tools** is still in preview but is actively developed. The installation process is similar to any other extension in VS Code:

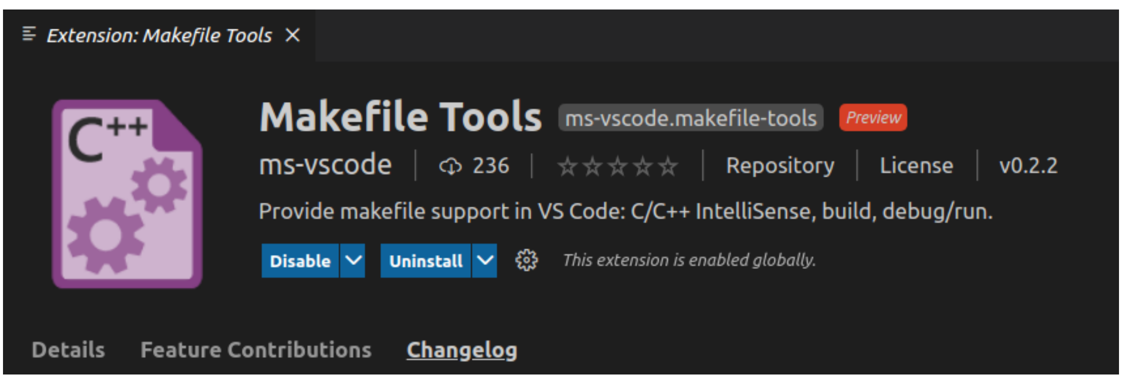


Figure 3: Makefile VSCode extension

After installing the extension, verify the availability of the `make` command in the system.

The most common implementation is GNU Make, which includes some non-standard extensions. If your installation of `make` is not available in the default path, you can configure it in VS Code at **File > Preferences > Settings > Extensions makefile**.

To compile and link the project, you can add a Makefile to the root of the project folder. It will be detected automatically by the extension. If you have a different structure, with a Makefile in another location, you can configure it at **File > Preferences > Settings > Extensions > makefile**.

This sample Makefile defines five simple rules:

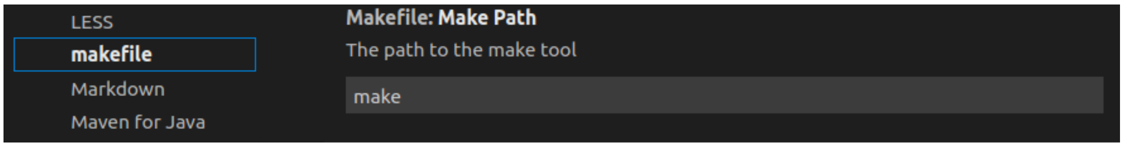


Figure 4: Make path

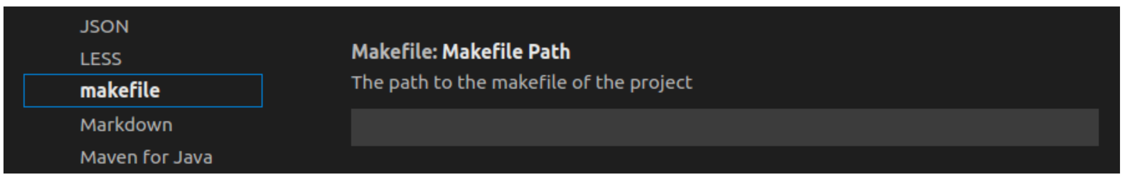


Figure 5: Makefile path

- `all`: Cleans the compiled files from the target folder, then compiles and run the test code.
- `default`: Delegates to `CoinFlipper.cpp`.
- `CoinFlipper.cpp`: Compiles the single source file.
- `test`: Delegates to `CoinFlipper.cpp`, then runs the output main function passing an argument.
- `clean`: Deletes compiled files.

```
#
# A simple makefile for compiling a c++ project
#
.DEFAULT_GOAL := CoinFlipper.cpp

all: clean test

CoinFlipper.cpp:
    gcc -o ./target/CoinFlipper.out ./src/main/CoinFlipper.cpp

run: CoinFlipper.cpp
    ./target/CoinFlipper.out 10

test: CoinFlipper.cpp
    ./target/CoinFlipper.out 10000

clean:
    rm -rf ./target/*.out
```

The Makefile Tools Extension provides a new "perspective" to the Visual Studio Code IDE. This contains three different commands and three different project configurations to run the Makefile:

The `Configuration:[Default]` refers to the make command configurations defined in the `.vscode/settings.json` file. This configuration is used to pass arguments to the make utility.

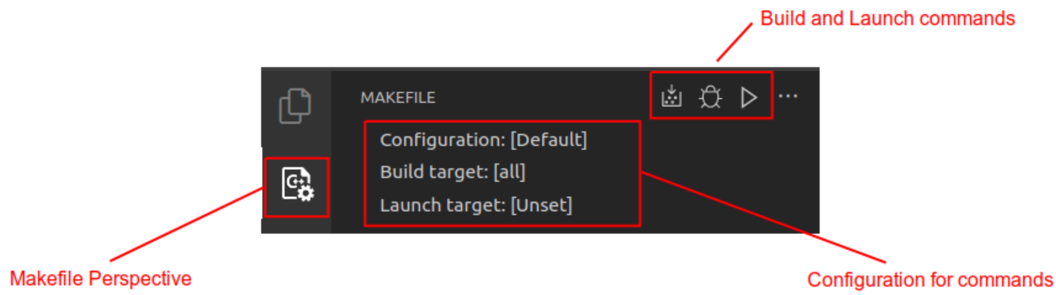


Figure 6: Makefile tools perspective

In the following example, two configurations are defined:

- `Default`
- `Print make version`

`Print make versions` adds the `--version` argument to the make utility every time the project is built using the Makefile extension. This argument is not especially useful but you can explore different arguments to fit your case.

```
{
 "makefile.configurations": [
     {
         "name": "Default",
         "makeArgs": []
     },
     {
         "name": "Print make version",
         "makeArgs": ["--version"]
     }
 ]
}
```

The second configuration is the default build target rule for the make utility, which is equivalent to running `make [target]` directly. The IDE will let show you a list of target rules defined in the Makefile configured for the project:

Finally, the third configuration available in the perspective is the `Launch target`. This shows you a list of compiled files that can be run from the perspective using the commands `Debug` and `Run`. This is useful if you want to debug your source code with GDB or LLDB debuggers.

In this example, the only file runnable is `CoinFlipper.out`, compiled from the source code.

The commands in the Makefile are self-explanatory:

- `Build` runs make with the target configured previously.
- `all` instead of `default` passes no arguments to the make utility.

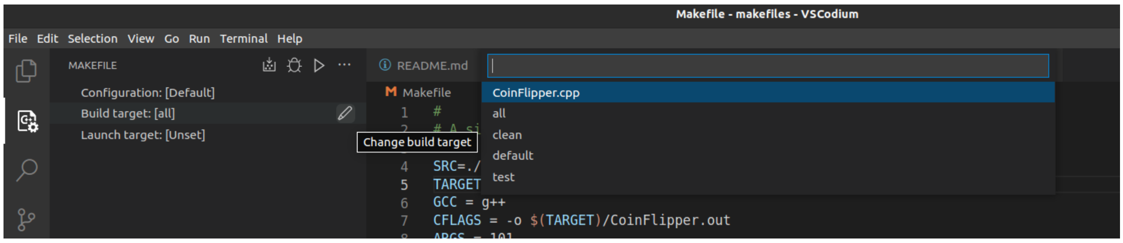


Figure 7: config build target

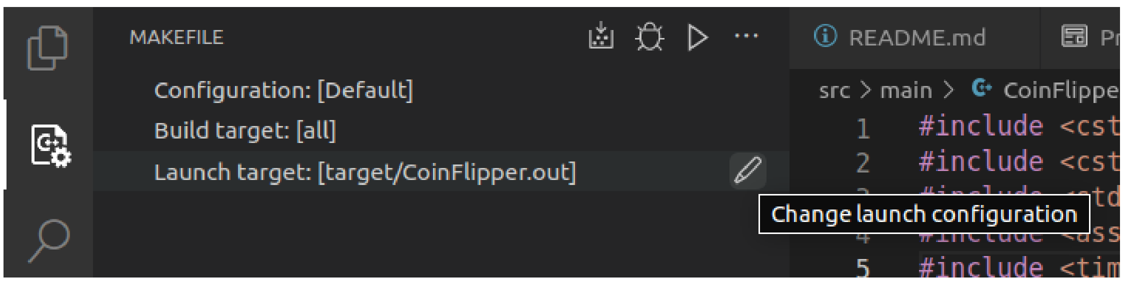


Figure 8: make launch target

- `Debug` and `Run in terminal` commands launch the target (`CoinFlipper.out` in the example) with/without the debug support.

Once you build the project, the terminal view shows the result of the execution:

```
Building target "all" with command: "make all"
rm -rf ./target/*.out
g++ -o ./target/CoinFlipper.out ./src/main/CoinFlipper.cpp
./target/CoinFlipper.out 101
50 Heads, 51 Tails
With enough trials Heads should be equal to Tails
Target all built successfully.
```

As you can see from the previous image, the target was built successfully after cleaning, compiling, and running the compiled program. The extension also provides commands to run other targets easily without changing the configurations in the perspective.

The following image shows the commands available for the Makefile in the sample project:

Building Complex Projects

Makefiles are more complex than this. Many projects have several levels of dependencies, configurations, and quirks that make supports easily. For example, the FFmpeg project is a collection of libraries to work with audio, video, and subtitles among other utilities. To build it, you can download the source from GitHub and examine the Makefile:

The developer documentation for the project states that before building the source code with the

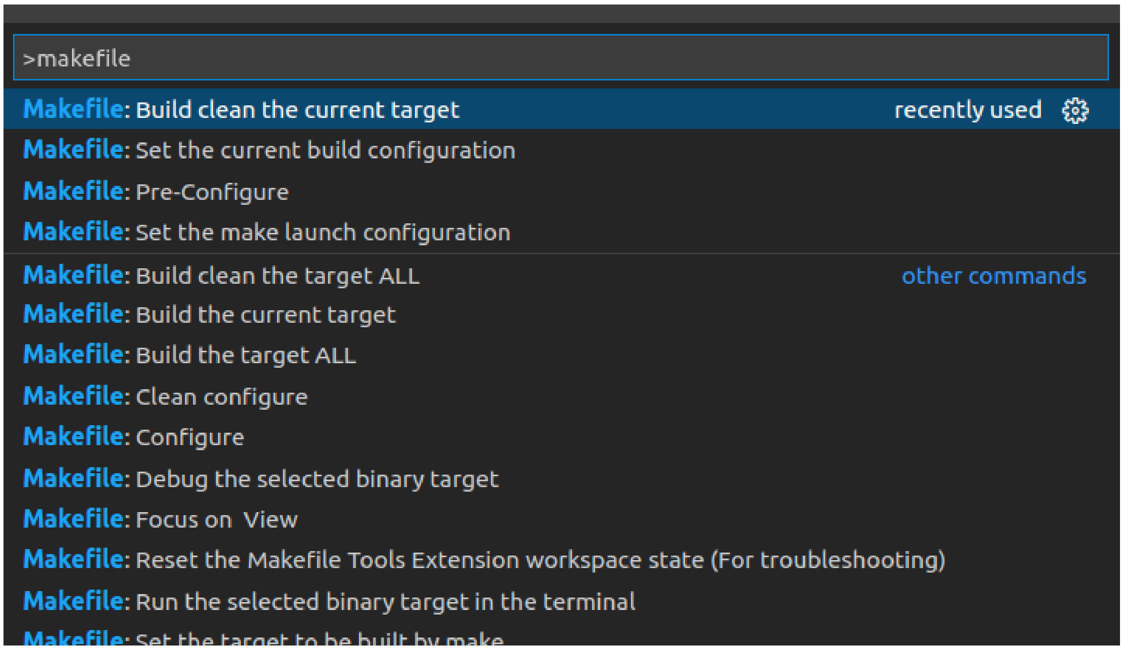


Figure 9: The makefile commands palette.

provided Makefile, you need to run the `configure` script located at the root of the project's source code. Fortunately, the Makefile Tools Extension provides a setting to define the preconfiguration files required to run before executing the make commands, again in **File > Preferences > Settings**.

In the **Commands** section of the Makefile Tools Extension perspective, you can run the preconfigure command. This will run the configure script, and then you're ready to experiment with the Makefile through the extension.

Conclusion

Large codebases need a build system to keep them under the development team's control, and Makefiles are one the most ubiquitous and flexible ways to define building these complex software projects.

With the new Makefile Tools Extension, Visual Studio Code greatly simplifies access for new developers. Though it is still tagged as in preview, this extension has been thoroughly tested by the Microsoft Team, building over seventy open-source projects written in different languages (including C, C++, and Python) successfully.

```
Makefile
MAIN_MAKEFILE=1
include ffbuild/config.mak

vpath %.c    $(SRC_PATH)
vpath %.cpp  $(SRC_PATH)
vpath %.h    $(SRC_PATH)
vpath %.inc  $(SRC_PATH)
vpath %.m    $(SRC_PATH)
vpath %.S    $(SRC_PATH)
vpath %.asm  $(SRC_PATH)
vpath %.rc   $(SRC_PATH)
vpath %.v    $(SRC_PATH)
vpath %.texi $(SRC_PATH)
vpath %.cu   $(SRC_PATH)
vpath %.ptx  $(SRC_PATH)
vpath %/fate_config.sh.template $(SRC_PATH)

TESTTOOLS   = audiogen videogen rotozoom tiny_psnr tiny_ssim base64 audiomatch
HOSTPROGS  := $(TESTTOOLS:%=tests/%) doc/print_options

# $(FFLIBS-yes) needs to be in linking order
FFLIBS-$(CONFIG_AVDEVICE)   += avdevice
FFLIBS-$(CONFIG_AVFILTER)   += avfilter
FFLIBS-$(CONFIG_AVFORMAT)   += avformat
FFLIBS-$(CONFIG_AVCODEC)    += avcodec
FFLIBS-$(CONFIG_POSTPROC)   += postproc
FFLIBS-$(CONFIG_SWRESAMPLE) += swresample
FFLIBS-$(CONFIG_SWSCALE)    += swscale

FFLIBS := avutil

DATA_FILES := $(wildcard $(SRC_PATH)/presets/*.ffpreset) $(SRC_PATH)/doc/ffprobe.xsd

SKIPHEADERS = compat/w32pthreads.h

# first so "all" becomes default target
all: all-yes

include $(SRC_PATH)/tools/Makefile
```

Figure 10: The makefile for FFmpeg.

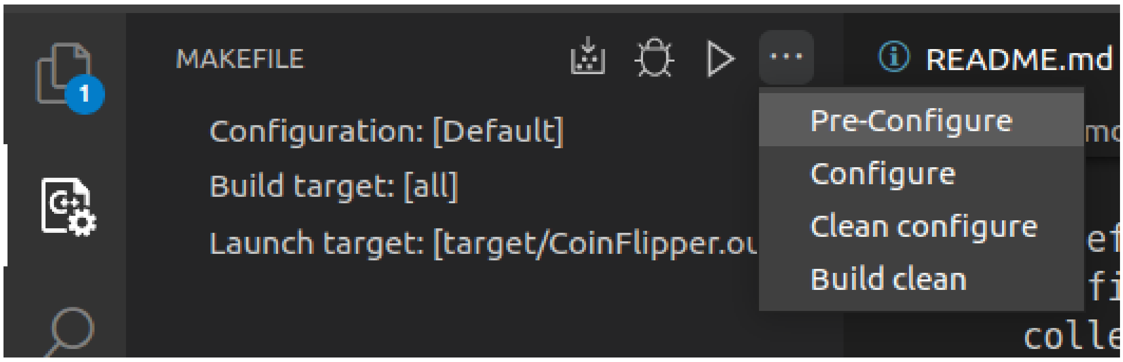


Figure 11: The makefile preconfiguration.

Understanding and Using Makefile Flags

`make` is a commonplace utility in the development world. It automates the process of generating executables, documentations, and other non-source files from the source code by dividing the build process into separate interrelated steps. Using `make` eliminates the need for typing out long and complex commands to compile the source code. `make` also compiles only the modified files, thereby saving time and processing resources.

Usually, the build process involves invoking various command-line tools, like the compiler or preprocessor. Often you need to pass options to these tools as per your requirements. However, hard-coding these options in your `makefile` can lead to difficulties. As an example, consider the following `makefile` snippet:

```
main.o: main.c
    gcc -Wall -c main.c
```

This snippet compiles `main.c` to `main.o` by invoking `gcc` with the `-Wall` option. But let's say that you do not wish to pass the `-Wall` option; instead, you want to pass the `-Werror` option. The only way of doing that is to edit the `makefile` to change the options. There is no convenient way to override the options without modifying the `makefile`. This is where `make` flags come into play.

Flags in `make` are just variables containing options that should be passed to the tools used in the compilation process. Although you can use any variable for this purpose, `make` defines some commonly used flags with default values for some common tools, including C and C++ compiler, C preprocessor, `lex`, and `yacc`. For example, `CFLAGS` is used to pass options to the C compiler, while `CXXFLAGS` is used in conjunction with the C++ compiler.

Why Should You Use Flags?

There are a few benefits to using flags over hard-coded options.

First, just like any other `makefile` variable, these flags can be overridden when invoking `make` from the command line. This feature offers a way to use any flag the user desires, as well as provides a default. For example, consider the following `makefile`:

```
CFLAGS = -g

all: main.o
    gcc -o main $(CFLAGS) main.o
```

When you run `make`, it executes `gcc -o main -g main.o`. The value of `$(CFLAGS)` is substituted when the command is executed. However, you can change the value of `$(CFLAGS)` by providing the new value when invoking `make`:

```
make CFLAGS="-Wall"
```

This time, the command that will be executed is `gcc -o main -Wall main.o`. The value of `$(CFLAGS)` provided in the command line overrides the defined value in the `makefile`.

Since any `make` variable can be overridden by providing its value in the command line, you may wonder why the manual recommends using special names for the variables. The reason is that by using the flags you can make use of the implicit rules provided by `make`. The implicit rules are a list of built-in rules that utilize the flags. For example, consider the following `makefile`:

```
CC = gcc
CFLAGS = -g # Flag to pass to gcc
CPPFLAGS = -I. # Flag to pass to the C preprocessor

all: main.o
```

If you have a `main.c` file in the project directory, running `make` will automatically compile it to `main.o`, even though you did not explicitly add any coding to build `main.o`. This is because `make` uses a built-in rule of the form `$(CC) $(CPPFLAGS) $(CFLAGS) -c -o x.o x.c` to compile any C file `x.c` into `x.o`. Thus, using implicit rules, you don't have to explicitly write the coding.

Another reason is that these flags are standardized and have been used for a long time, so anyone building your software will expect you to use these flags. Using any other variable would force them to go through your `makefile` in order to figure out which variable is being used. Instead, by sticking to the standard, you can save them time.

How to Use `Make` Flags

You can use `make` flags just like any other `make` variable. Define the flags with default values using the `=` operator, and use the flags using the `$(...)` syntax:

```
CC = gcc # It is a recommended practice to define the C compiler with CC
CFLAGS = -Wall # Defines -Wall as default flag

main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

You can also override the flags when invoking main, as explained earlier:

```
make CFLAGS="-g -Wall"
```

Since `make` already defines these flags with default values (an empty string for most of them), you don't have to define them in the `makefile` explicitly if you don't want to have a default value, and you can use them directly from the command line. For example, the following `makefile` is valid, and `CFLAGS` is set to the empty string, which means no options are passed to the compiler.

```
CC = gcc # It is a recommended practice to define the C compiler with CC

main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

You can still define `CFLAGS` from the command line:

```
make CFLAGS="-Wall"
```

Some Commonly Used Flags

Here are a few commonly used flags. For a full list of flags, check the manual.

CFLAGS

This flag should contain the options to give to the C compiler. These options can include debug options, optimization level, warning levels, and any extra flags that you want to use.

```
CC = gcc

CFLAGS = -g -Wall # Passes -g and -Wall to gcc

main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

If you have options that are required for proper compilation, the manual suggests putting the optional ones in `CFLAGS` and adding the required options to `CFLAGS` separately. This way the user can override `CFLAGS` via the command line, but the required options will not be overridden.

```
CFLAGS = -g # Optional. Not required for proper compilation
ALL_CFLAGS = -I. $(CFLAGS) # -I. is required for proper compilation
main.o: main.c
        $(CC) -c $(ALL_CFLAGS) main.c
```

CXXFLAGS

This flag is similar to `CFLAGS`, except that you should use `CXXFLAGS` when invoking a C++ compiler.

```
CXX = g++

CXXFLAGS = -g -Wall # Passes -g and -Wall to g++

main.o: main.cpp
    $(CXX) $(CXXFLAGS) -o main.o main.cpp
```

CPPFLAGS

`CPPFLAGS` is used to pass extra flags to the C preprocessor. These flags are also used by any programs that use the C preprocessor, including the C, C++, and Fortran compilers. You do not need to explicitly call the C preprocessor. Pass `CPPFLAGS` to the compiler, and these will be used when the compiler invokes the preprocessor. The most common use case of `CPPFLAGS` is to include directories to the compiler search path using the `-I` option.

```
CC = gcc
CFLAGS = -g -Wall
CPPFLAGS = - I /usr/foo/bar # Search for header files in /usr/foo/bar

main.o: main.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c main.c
```

LDFLAGS

You can use `LDFLAGS` to pass extra flags to the linker `lD`. Similar to `CPPFLAGS`, these flags are automatically passed to the linker when the compiler invokes it. The most common use is to specify directories where the libraries can be found, using the `-L` option. You should not include the names of the libraries in `LDFLAGS`; instead they go into `LDLIBS`.

```
LDFLAGS = -L. \ # Search for libraries in the current directory
          -L/usr/foo # Search for libraries in /usr/foo

main.o: main.c
    gcc $(LDFLAGS) -c main.c
```

LDLIBS

The `LDLIBS` flag should contain the space-separated list of libraries that are used by your programs. For this flag, the `-l` option followed by the name of the library is used. For example, if your software uses `libm`, the math library, then you need to include the `-lm` option.

```
LDFLAGS = -L. \ # Search for libraries in the current directory
          -L/usr/foo # Search for libraries in /usr/foo

LDLIBS = -lm -lfoo # Use libm and libfoo

main.o: main.c
    gcc $(LDFLAGS) -c main.c $(LDLIBS)
```

Keep in mind that `LDLIBS` should be included *after* you have listed all your source files. Otherwise the linker will not be able to link the symbols properly.

LFLAGS

This flag is used if you are working with `lex`, a tool used to generate lexical analyzers. `Lex` takes a list of token definitions in a `.l` file and generates a C program that can take an input and tokenize it accordingly. You can find a basic introduction to `lex` on IBM's documentation site.

```
LEX = flex # Use flex as the lex program
LFLAGS = -d # enable debug

lexer.c: lexer.l
    $(LEX) $(LFLAGS) lexer.l
```

YFLAGS

This flag is used to pass options to `yacc`. This is a tool that is often used in conjunction with `lex`. `Yacc` is a parser generator; it converts a grammar definition in a `.y` file into a C program, which can parse the tokenized output of `lex` into a parse tree. IBM has a tutorial if you'd like to find out more about `yacc`.

```
YACC = bison # Use bison as the yacc program
YFLAGS = -v \ # Verbose mode
        -g # Generate graph

parser.c: parser.y
    $(YACC) $(YFLAGS) parser.y
```

MAKEFLAGS

This is an interesting flag that is used in recursive invocation of `make`. If you have modules or subsystems in your project, it is likely that each subsystem will have its own `makefile`. The top-level `makefile` will then recursively call `make` for each of the modules. The `MAKEFLAGS` variable is automatically set up by `make`, and it contains all the flags and command line variables that you passed to the top-level `make`. The `MAKEFLAGS` variables will pass these options and variables down to each sub-`make`.

To test this, create a directory called `subdir` and create a `makefile` in this `subdir` with the following content:

```
all:
 echo $(MAKEFLAGS)
```

This will print out the value of the `MAKEFLAGS` variable.

Then in your top-level `makefile`, write the following:

```
subsystem:
 cd subdir && $(MAKE)
```

This `makefile` recursively calls `make` in the `subdir` subdirectory.

Now you can run `make` from your project root with options:

```
$  make -sk CFLAGS="-g"
ks -- CFLAGS=-g
```

As you can see, the options `-k` and `-s` were passed to the sub-`make`, as well as the variables.

Note that the options `-C`, `-f`, `-o`, and `-W` are not put into `MAKEFLAGS` and not passed down. You can read more about `MAKEFLAGS` on GNU.org.

Conclusion

Using `make` flags ensures your `makefile` follows the standard and offers an easy and powerful way to customize the behaviors of the compilation tools by providing them options. However, `make` flags are limited and require a deep understanding of the right tools to use.

Understanding and Using Makefile Variables

Since its appearance in 1976, Make has been helping developers automate complex processes for compiling code, building executables, and generating documentation.

Like other programming languages, Make lets you define and use variables that facilitate reusability of values.

Have you found yourself using the same value in multiple places? This is both repetitive and prone to errors. If you'd like to change this value, you'll have to change it everywhere. This process is tedious, but it can be solved with variables, and Make offers powerful variable manipulation techniques that can make your life easier.

In this chapter, you'll learn all about `make` variables and how to use them.

What Are Make Variables?

A variable is a named construct that can hold a value that can be reused in the program. It is defined by writing a name followed by `=`, `:=`, or `::=`, and then a value. The name of a variable can be any sequence of characters except ":", "#", "=", or white space. In addition, variable names in Make are case sensitive, like many other programming languages.

The following is an example of a variable definition:

```
foo = World
```

Any white space before the variable's value is stripped away, but white spaces at the end are preserved. Using a `$` inside the value of the variable is permitted, but `make` will assume that a string starting with the `$` sign is referring to another variable and will substitute the variable's value:

```
foo = one$two
# foo becomes onewo
```

As you'll soon learn, `make` assumes that `$t` refers to another variable named `t` and substitutes it. Since `t` doesn't exist, it's empty, and therefore, `foo` becomes `onewo`. If you want to include a `$` verbatim, you must escape it with another `$`:

```
foo = one$$two
```

How to Use Make Variables

Once defined, a variable can be used in any target, prerequisite, or recipe. To substitute a variable's value, you need to use a dollar sign (`$`) followed by the variable's name in parentheses or curly braces. For instance, you can refer to the `foo` variable using both `${foo}` and `$(foo)`.

Here's an example of a variable reference in a recipe:

```
foo = World
all:
    echo "Hello, $(foo)!"
```

Running `make` with the earlier `makefile` will print "Hello, World!".

Another common example of variable usage is in compiling a C program where you can define an `objects` variable to hold the list of all object files:

```
objects = main.o foo.o bar.o
program : $(objects) # objects used in prerequisite
    cc -o program $(objects) # objects used in recipe

$(objects) : foo.h # objects used in target
```

Here, the `objects` variable has been used in a target, prerequisite, and recipe.

Unlike many other programming languages, using a variable that you have not set explicitly will *not* result in an error; rather, the variable will have an empty string as its default value. However, some special variables have built-in non-empty values, and several other variables have different default values set for each different rule (more on this later).

How to Set Variables

How to Set Variables

Setting a variable refers to defining a variable with an initial value as well as changing its value later in the program. You can either set a value explicitly in the `makefile` or pass it as an environment variable or a command-line argument.

Variables in the Makefile

There are four different ways you can define a variable in the Makefile:

- Recursive assignment
- Simple assignment
- Immediate assignment
- Conditional assignment

Recursive and Simple Assignment As you may remember, you can define a variable with `=`, `:=`, and `::=`. There's a subtle difference in how variables are expanded based on what operator is used to define them.

- The variables defined using `=` are called recursively expanded variables, and
- Those defined with `:=` and `::=` are called simply expanded variables.

When a recursively expanded variable is expanded, its value is substituted verbatim. If the substituted text contains references to other variables, they are also substituted until no further variable reference is encountered. Consider the following example where `foo` expands to `Hello $(bar)`:

```
foo = Hello $(bar)
bar = World
```

```
all:
    @echo "$(foo)"
```

Since `foo` is a recursively expanded variable, `$(bar)` is also expanded, and "Hello World" is printed. This **recursive expansion** process is performed every time the variable is expanded, using the *current values* of any referenced variables:

```
bar = World
foo = Hello $(bar)

bar = Make
# foo now expands to "Hello Make"

all:
    @echo ${foo} # prints Hello Make
```

The biggest advantage of recursively expanded variables is that they make it easy to construct new variables piecewise: you can define separate pieces of the variable and string them together. You can define more granular variables and join them together, which gives you finer control over how `make` is executed.

For example, consider the following snippet that is often used in compiling C programs:

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
main.o: main.c
    $(CC) -c $(ALL_CFLAGS) main.c
```

Here, `ALL_CFLAGS` is a recursively expanded variable that expands to include the contents of `CFLAGS` along with the `-I.` option. This lets you override the `CFLAGS` variable if you wish to pass other options while retaining the mandatory `-I.` option:

```
CFLAGS="-g -Wall" # ALL_CFLAGS expands to "-I. -g -Wall"
```

A disadvantage of recursively expanded variables is that it's not possible to append something to the end of the variable:

```
CFLAGS = $(CFLAGS) -I. # Causes infinite recursion
```

To overcome this issue, GNU Make supports another flavor of variable known as **simply expanded variables**, which are defined with `:=` or `::=`. A simply expanded variable, when defined, is scanned for further variable references, and they are substituted once and for all.

Unlike recursively expanded variables, where referenced variables are expanded to their current values, in a simply expanded variable, referenced variables are expanded to their values at the time the variable is defined:

```
bar := World
foo := Hello $(bar)

bar = Make
```

```
all:
	@echo ${foo} # Prints Hello World
```

With a simply expanded variable, the following is possible:

```
CFLAGS = $(CFLAGS) -I.
```

GNU Make supports simply and recursively expanded variables. However, other versions of `make` usually only support recursively expanded variables. The support for simply expanded variables was added to the Portable Operating System Interface (POSIX) standard in 2012 with only the `::=` operator.

Immediate Assignment A variable defined with `:::=` is called an **immediately expanded variable**. Like a simply expanded variable, its value is expanded immediately when it's defined. But like a recursively expanded variable, it will be re-expanded every time it's used. After the value is immediately expanded, it will automatically be quoted, and all instances of `$` in the value after expansion will be converted into `$$`.

In the following code, the immediately expanded variable `foo` behaves similarly to a simply expanded variable:

```
bar := World
foo :::= Hello $(bar)

bar = Make

all:
	@echo ${foo} # Prints Hello World
```

However, if there are references to other variables, things get interesting:

```
var = one$$two
OUT :::= $(var)
var = three$$four
```

Here, `OUT` will have the value `one$$two`. This is because `$(var)` is immediately expanded to `one$two`, which is quoted to get `one$$two`. But `OUT` is a recursive variable, so when it's used, `$two` will be expanded:

```
two = two

all:
	@echo ${OUT} # onetwo
```

The `:::=` operator is supported in POSIX Make, but GNU Make includes this operator from version 4.4 onward.

Conditional Assignment The conditional assignment operator `?=` can be used to set a variable only if it hasn't already been defined:

```
foo = World
```

```
foo ?= Make # foo will not change
bar ?= Make # bar will change

all:
	@echo Hello ${foo}
	@echo Hello ${bar}
```

An equivalent way of defining variables conditionally is to use the **origin** function:

```
foo ?= Make

# is equivalent to

ifeq ($(origin foo), undefined)
foo = Make
endif
```

These four types of assignments can be used in some specific situations:

Shell Assignment

You may sometimes need to run a shell command and assign its output to a variable. You can do that with the **shell** function:

```
files = $(shell ls) # Runs the `ls` command & assigns its output to `files`
```

A shorthand for this is the shell assignment operator **!=**. With this operator, the right-hand side must be the shell command whose result will be assigned to the left-hand side:

```
files != ls
```

Variables With Spaces

Trailing spaces at the end of a variable definition are preserved in the variable value, but spaces at the beginning are stripped away:

```
foo = xyz   # There are spaces at the beginning and at the end

# Prints "startxyz   end"
all:
	@echo "start${foo}end"
```

It's possible to preserve spaces at the beginning by using a second variable to store the space character:

```
nullstring =
foo = ${nullstring} xyz   # Spaces at the end

# Prints "start xyz   end"
all:
	@echo "start${foo}end"
```

Target-Specific Variables

It's possible to limit the scope of a variable to specific targets only. The syntax for this is as follows:

```
target ... : variable-assignment
```

Here's an example:

```
target-one: foo = World
target-two: foo = Make

target-one:
    @echo Hello ${foo}

target-two:
    @echo Hello ${foo}
```

Here, the variable `foo` will have different values based on which target `make` is currently evaluating:

```
$ make target-one
Hello World

$ make target-two
Hello Make
```

Pattern-Specific Variables

Pattern-Specific Variables

Pattern-specific variables make it possible to limit the scope of a variable to targets that match a particular pattern. The syntax is similar to target-specific variables:

```
pattern ... : variable-assignment
```

For example, the following line sets the variable `foo` to `World` for any target that ends in `.c`:

```
%.c: foo = World
```

Pattern-specific variables are commonly used when you want to set the variable for **multiple targets that share a common pattern**, such as setting the same compiler options for all C files.

Environment Variables

The real power of `make` variables starts to show when you pair them with environment variables. When `make` is run in a shell, any environment variable present in the shell is transformed into a `make` variable with the same name and value. This means you don't have to set them in the `makefile` explicitly:

```
all:
    @echo ${USER}
```

When you run the earlier `makefile`, it should print your username since the `USER` environment variable is present in the shell.

This feature is most commonly used with flags. For example, if you set the `CFLAGS` environment variable with your preferred C compiler options, they will be used by most `makefiles` to compile C code since, conventionally, the `CFLAGS` variable is only used for this purpose. However, this is only sometimes guaranteed, as you'll see next.

If there's an explicit assignment in the `makefile` to a variable, it overrides any environment variable with the same name:

```
USER = Bob

all:
    @echo ${USER}
```

The earlier `makefile` will always print `Bob` since the assignment overrides the `$USER` environment variable. You can pass the `-e` flag to `make` so environment variables override assignments instead, but this is not recommended, as it can lead to unexpected results.

Command-Line Arguments

You can pass variable values to the `make` command as command-line variables. Unlike environment variables, command-line arguments will always override assignments in the `makefile` unless the `override` directive is used:

```
override FOO = Hello
BAR = World

all:
    @echo "${FOO} ${BAR}"
```

You can simply run `make`, and the default values will be used:

```
$ make
Hello World
```

You can pass a new value for `BAR` by passing it as a command-line argument:

```
$ make BAR=Make
Hello Make
```

However, since the `override` directive is used with `FOO`, it cannot be changed via command-line arguments:

```
$ make FOO=Hi
Hello World
```

This feature is handy since it lets you change a variable's value without editing the `makefile`. This is most commonly used to pass configuration options that may vary from system to system or used to customize the software. As a practical example, Vim uses command-line arguments to override configuration options, like the runtime directory and location of the default configuration.

How To Append To a Variable

How to Append to a Variable

You can use the previous value of a simply expanded variable to add more text to it:

```
foo := Hello
foo := ${foo} World

# prints "Hello World"
all:
    @echo ${foo}
```

As mentioned before, this syntax will produce an **infinite recursion error** with a recursively expanded variable. In this case, you can use the `+=` operator, which appends text to a variable, and it can be used for both recursively expanded and simply expanded variables:

```
foo = Hello
foo += World

bar := Hello
bar += World

# Both print "Hello World"
all:
    @echo ${foo}
    @echo ${bar}
```

However, there's a subtle difference in the way it works for the two different flavors of variables, which you can read about in the docs.

How To Use Special Variables

In Make, any variable that is not defined is assigned an *empty string* as the default value. There are, however, a few special variables that are exceptions:

Automatic Variables

Automatic variables are special variables whose value is set up automatically per rule based on the target and prerequisites of that particular rule. The following are several commonly used automatic variables:

- `$@` is the file name of the target of the rule.
- `$<` is the name of the first prerequisite.
- `$?` is the name of all the prerequisites that are newer than the target, with spaces between them. If the target does not exist, all prerequisites will be included.
- `$^` is the name of all the prerequisites, with spaces between them.

Here's an example that shows automatic variables in action:

```
hello: one two
    @echo $@
    @echo $<
    @echo $?
    @echo $^

    @touch hello

one:
    @touch one

two:
    @touch two

clean:
    @rm -f hello one two
```

Running `make` with the earlier `makefile` prints the following:

```
hello
one
one two
one two
```

If you run `touch one` to modify `one` and run `make` again, you'll get a different output:

```
hello
one
one
one two
```

Since `one` is newer than the target `hello`, `$?` contains only `one`.

There exist variants of these automatic variables that can extract the directory and file-within-directory name from the matched expression. You can find a list of all automatic variables in the official docs.

Automatic variables are often used where the target and prerequisite names dictate how the recipe executes. A very common practical example is the following rule that compiles a C file of the form `x.c` into `x.o`:

```
%.o:%.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
```

Implicit Variables

Make ships with certain predefined rules for some commonly performed operations. These rules include the following:

- Compiling `x.c` to `x.o` with a rule of the form `$(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@`

- Compiling `x.cc` or `x.cpp` with a rule of the form `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@`
- Linking a static object file `x.o` to create `x` with a rule of the form `$(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)`
- And many more

These implicit rules make use of certain predefined variables known as implicit variables. Some of these are as follows:

- **`CC`** is a program for compiling C programs. The default is `cc`.
- **`CXX`** is a program for compiling C++ programs. The default is `g++`.
- **`CPP`** is a program for running the C preprocessor. The default is `$(CC) -E`.
- **`LEX`** is a program to compile Lex grammars into source code. The default is `lex`.
- **`YACC`** is a program to compile Yacc grammars into source code. The default is `yacc`.

You can find the full list of implicit variables in GNU Make's docs.

Just like standard variables, you can explicitly define an implicit variable:

```
CC = clang

# This implicit rule will use clang as compiler
foo.o:foo.c
```

Or you can define them with command line arguments:

```
make CC=clang
```

Flags

Flags are special variables commonly used to pass options to various command-line tools, like compilers or preprocessors. Compilers and preprocessors are implicitly defined variables for some commonly used tools, including the following:

- **`CFLAGS`** is passed to `CC` for compiling C.
- **`CPPFLAGS`** is passed to `CPP` for preprocessing C programs.
- **`CXXFLAGS`** is passed to `CXX` for compiling C++.

Learn more about Makefile flags.

Conclusion

Variables in Make are similar to variables in other programming languages. However, certain features and quirks make them powerful and convenient to use, albeit slightly difficult to wrap your head around. This chapter gave you an overview of the different types of variables in Make and how you can use them.

Using Makefile Wildcards

Although many of the new modern programming frameworks, like Node.js and .NET, come with their own way of packaging and distributing their programs, there's no doubt that Make originally created a lot of the founding principles for building and distributing software.

Make provides users with many exciting possibilities, including making packaging software easy and automated. This saves time when building software, and it's a massive aid in creating a streamlined process. Once you get started building Makefiles, you'll notice that there are places where you don't want something to be hardcoded. This is where wildcards come into play. They're one of the parts that turn Make into an incredibly flexible tool build tool.

In this chapter, you'll get a quick introduction to Make, where you'll be shown an example C application. Don't be worried if you're not familiar with C programming; the application is simple to understand, and your familiarity with any language is more than enough. With this application, you'll be guided through various ways to implement wildcards into the build process.

If you want to see all the code from this tutorial in one place, you can find it in this GitHub repository.

How To Use Make

How to use make

To begin, define a sample application you can use as an example. As any experienced programmer will know, the best example is that of "Hello, World!" This is what it looks like in C:

```
#include <stdio.h> // Include the library necessary to print to the terminal

int main() {
   // printf() displays the string inside quotation
   printf("Hello, World!");
   return 0; // Make sure the program terminates after completion
}
```

The program is relatively simple, and if any line is confusing, you can look at the accompanying comment.

Copy the contents of the code block and save it in a file called `main.c`. Now create a file called `Makefile` in the same directory as your `main.c` file, and paste the following into it:

```
main.o: main.c
  gcc -o hello main.c
```

> Note: Make is very particular about indentation, so make sure you use a `tab` on the second line.

It's assumed that you are familiar with Make and its syntax, but you may be unfamiliar with GCC. It's the compiler most commonly used for C programs. In this command, you define that `gcc` should compile the program into a binary called `hello`, and it should do this using the `main.c` file.

Now, the basis of the application is done, and it's time to introduce wildcards.

Makefile Wildcards

Makefile Wildcard

As mentioned in the introduction, when you want your Makefile targets to be flexible, wildcards come into play. Wildcards can be effective in many places but only pick up files matching a pattern. Now, dive deeper into what is possible with wildcards:

Common Wildcard Use

If you've worked in a terminal before or with glob patterns, you may be familiar with an asterisk (`*`) being used to match any character. This is also how wildcards work in Make. For instance, you can use `*.o` to match any files with the extension `.o`.

You'll often use a wildcard character to make a `clean` target. Earlier we generated a main.o file and it's certainly possible to manually remove it and any other generated files, but you'll see almost all projects using Make contain a `clean` target. This target could look something like this:

```
clean:
  rm -f *.o
```

Running `make clean` will ensure that any files ending in `.o` will get deleted, helping you keep a clean directory.

Wildcard Function

As you can see, the use of wildcards is not as complex as it may seem on the surface. If you've ever worked with string matching, the wildcard function will seem very familiar.

However, an important thing to note is that you can't do wildcard matching everywhere inside a `Makefile`, at least not in the way shown earlier. For instance, take a look at the following example:

```
files_to_delete = *.o

clean:
  rm -f $(files_to_delete)
```

This will work fine because the `rm` command takes the argument `*.o` and can work with it. But it's important to note that the command that's being run is `rm -f *.o` and that `$(files_to_delete)` is *not* replaced with a list of files matching `*.o`. So while many commands invoked in a Makefile may work fine by directly inserting `*.o`, it's important to know the distinction.

For instance, what you see inside a recipe is only evaluated *once*, not recursively. Imagine that `make` is reading every line from left to right. It will encounter the variable `files_to_delete`, and then replace it with the contents of the variable; `*.o`. At the end of that variable, it continues moving to the right. It's not reading over the line again to find out that there's a wildcard that needs to be expanded. This is why you need to define the wildcard directly in the recipe works, as the wildcard is what is now encountered when reading the line.

If you want the variable to contain the actual list of files, you have to make a slight variation and use the `wildcard` function, like so:

```
files_to_delete = $(wildcard *.o)

clean:
  rm -f $(files_to_delete)
```

This is one of the most common pitfalls in `make`. Now `make` will read the variable first and evaluate the `wildcard` function, meaning the variable actually contains the list of files. Then, when the variable is called in the recipe, it's a list of files.

Rules With Wildcards

Rules Wildcard

You've now seen some examples of how wildcards can be used inside of Makefiles, but it's also possible to use pattern matching when defining your rules. By defining a rule inside your Makefile with the `%` character, you can refer to the pattern inside the target by using the character sequence `$*`. As an example, here's how you can integrate a wildcard into a rule where you want to create a binary from a given `.c` file:

```
%.out: %.c
  gcc -o $* $*.c
```

Now you can run `make main.out`, and it will create the `main` binary from the `main.c` file.

Associated Functions

You've now learned about most of the uses that are specific to Make, but it's important to note that there are also places inside Make where you can use wildcards like you would in many other scenarios you're used to, like Bash programming. Here are a few examples:

The Patsubst Function The `patsubst` function inside Make is a useful one, giving you the ability to modify strings based on a pattern. The functionality in itself is very basic; it finds some text and replaces it. The syntax is as follows:

```
$(patsubst pattern,replacement,text)
```

A straightforward example of using this function could be `$(patsubst world,everyone,hello world)`, which would produce the text to "hello everyone". From here, you can search for any pattern using the `%` character and get it replaced, like so:

```
$(patsubst he%,%x,hello world)
```

The previous code produces the text "llox world" because you've dropped `he` and added an `x`. This function is not a string replacement tool; it's a pattern replacement tool.

Filter Just like the `patsubst` function, `filter` is a text-manipulation function. You use the `filter` function when you want to return a list of words that match a given pattern, and the syntax for this command is `$(filter pattern...,text)`.

As you can see, it's possible to specify many different patterns you want to match. Here's an example:

```
files = foo.c bar.c foo.o bar.o

foo:
  cat $(filter %.c, $(files))
```

In this example, the relevant projects files are `foo.c bar.c foo.o bar.o`, but using this rule, you only want to know the contents of the files with the extension `.c`.

Conclusion

Wildcards are a handy utility when creating your Makefiles. You can use them directly in your rules, however, you have to ensure that you're using them correctly and consider whether you need to use the `wildcard` function. Besides by using the wildcards directly in your targets, you can also use pattern matching in your rules to create more dynamic targets.

Autotools is one of the most widely adopted code packaging and shipping tools available to developers on Linux. While there are alternatives, such as CMake, SCons, and BJam, they don't quite match Autotools in ease of use, power, and versatility.

At its base, Autotools can help make your application more portable, give it the versatility to be installed on many different systems, and can automatically procure scripts to check where elements are, like the compiler for your program. In this chapter, you will learn how to use Autotools to package up an application and ship it.

Components of Autotools

Autotools is made up of three unique components:

- `autoconf`
- `automake`
- `aclocal`

Using these tools, you will create two files, `configure` and `Makefile.in`. These files are present in any project shipped using Autotools and are usually quite large and complex. Luckily, you don't have to write them yourself—instead you'll be writing the files `configure.ac` and `Makefile.am`, which will automatically generate the files you need.

Autoconf

Autoconf is written in M4sh, using `m4` macros. If you've heard of this before, then great! If you haven't, no worries. `m4sh` provides some macros you can use when creating your `configure.ac` script and are part of why you can generate a massive `configure` script without having to write too much actual code.

The way it works is that you create a `configure.ac` script in which you define various settings like release name, version, which compiler to use, and where it should output files. Once you've written your script, you run it through `autoconf` to create your final `configure` script. The purpose of `autoconf` is to collect information from your system to populate the `Makefile.in` template, which is created using `automake`.

Automake

The `Makefile.am` script creates `Makefile.in`. The principles behind this are the same as with `configure.ac`: write a simple script in order to create a complex file. Automake is the component you'll use to create the Makefile, a template that can then be populated with `autoconf`.

Automake does so using variables and primaries. An example of such a primary is `bin_PROGRAMS = helloworld`, where the primary is the `_PROGRAMS` suffix. This primary gives `automake` some knowledge about your program, like where you want the produced binary to be installed.

In this case, you're telling `automake` to install the `helloworld` binary in the path defined by the `bindir`. You may notice we didn't define a `bindir` variable, because that variable is built into `automake` and is typically the default binary directory of your system.

Other examples of primaries are `_SCRIPTS`, which you can use when you want a script, rather than a binary to be installed somewhere, and `_DATA`, when you have extra data files you want included in your installation. There are many more that you will find once you start using Autotools and figure out what your needs are.

One last thing to mention is that although `Makefile.in` is special in that it contains all of these primaries, it's still a regular Makefile. Meaning you can write your own custom make targets if you want. For example, if you want to have a custom `clean` target that deletes specific files, you can do so easily.

`Aclocal`

This is the smallest component in the Autotools suite, but it's very important. You learned in the previous section that `autoconf` uses `m4` macros to be configured. But where do these `m4` macros come from? They're generated by running the `aclocal` command. Simple as that. If you don't run `aclocal` before running `autoconf`, you'll get an error complaining about missing macros.

Using Autotools

Now that you know the basic principles of how the Autotools suite is put together, it's time to see it all in action and create a small C program that you can compile and ship.

Writing the Source Program

The first thing you need is the program you want to compile and ship. Autotools is compatible with many different projects and languages, but for this example you'll be working in `C`, which is most commonly used. If you're not familiar with `C`, don't worry, it's very simple. Here's your sample code:

```c
#include <stdio.h>

int
main(int argc, char* argv[])
{
  printf("Hello World\n");
  return 0;
}
```

As you can see, it simply includes a standard in/standard out library and then prints `Hello World\n`. Let's start with`autoconf` to configure this project.

Configuring `configure.ac`

When writing your `configure.ac` file, there are a lot of options to choose from. You can get very specific about how you want your script to be configured, but some configurations need to be set. The first of these is `AC_INIT`. This tells `autoconf` what the name of your application is, what version it is, and who's the maintainer. For this example, you'll write:

```
AC_INIT([helloworld], [0.1], [maintainer@example.com])
```

While `autoconf` is generally used alongside `automake`, it's not necessary, so you need to initialize that by writing:

```
AM_INIT_AUTOMAKE
```

Now that the generic options are initialized, you can get more specific with what you want. You need to specify what compiler you want the `configure` script to use. You do this by writing:

```
AC_PROG_CC
```

This will tell the `configure` script to look for a `C` compiler. For other applications, you may need more dependencies to build your program. By using the `AC_PATH_PROG` macro, you can make `autoconf` look for specific programs in a user's `PATH`. At this point, there are only two steps needed to finish your basic `configure.ac` script:

```
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

`AC_CONFIG_FILES` tells `autoconf` that it should find a file called `Makefile.in` and replace placeholders according to what we've specified. This can be things like version or maintainer. `AC_OUTPUT` is the last thing you want to put in your `configure.ac` script, as it tells `autoconf` to output the final `configure` script. In the end, your `configure.ac` file should contain the following:

```
AC_INIT([helloworld], [0.1], [maintainer@example.com])
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Making the Makefile

When using `automake`, you'll have to adhere to a set of standards. One of these is that source files for a project are located in the `src` folder. In this project, you have a single `main.c` file in our root directory, so you need to tell `automake` that:

```
AUTOMAKE_OPTIONS = foreign
```

You need to tell `automake` what you want your compiled binary to be called. In this case, you want it to be called `helloworld`, so write the following:

```
bin_PROGRAMS = helloworld
```

Only one thing left, and that is to tell `automake` what files are needed to compile your application. Do this by writing:

```
helloworld_SOURCES = main.c
```

Notice how the first part is the name of your application followed by the `SOURCES` primary. Now `automake` knows all that it needs to know, and your `Makefile.am` is ready to use.

Creating Final Scripts

Once you've written your `configure.ac` and `Makefile.am`, it's relatively straightforward to distribute your application. Remember to start by running `aclocal` so you can run `autoconf`. Once you've run `autoconf`, you can run `automake --add-missing` to build your `Makefile.in`.

The reason for the `--add-missing` flag is to tell `automake` to automatically generate all of the additional files required, as usually you need more than just `Makefile.am` and would have to manually enter in the other files.

At this point, you have all you need to distribute your program. Before moving on on, here's a short recap showing the commands you should've run by now:

```
aclocal
autoconf
automake --add-missing
```

Distributing the Program

Distributing your application can seem like a daunting task, but Autotools makes it super easy. All you have to do is run `make dist` after you've run the configure scripts above. This will produce a tarball, which you can then ship to your customers.

Conclusion

Now you're able to use Autotools to compile and distribute your application, and you're able to do it in a way that ensures it's portable across a variety of systems. From here, you can start looking into automating this procedure and other ways to integrate Autotools directly into your daily development.

The great advantage of using something like Autotools is that you'll be using a system that has been in place for many years, is well-documented, and widely used. Many developers are comfortable installing applications using what Autotools produces, so it can make your application much more familiar and accessible.

Docker and Makefiles: Building and Pushing Images with Make

Deployments have been one of the hassles for many organizations for a long time, with companies sometimes even hiring engineers whose sole job is to get applications deployed more effectively. Because of this, many tools have been developed to help with this exact use case. However, some prefer to use tools that have already existed for many years: Docker and Makefiles.

As these are both very popular tools, it's likely that hearing they're commonly used isn't a surprise to many people, but the full extent to which they can be used together might be. Back when one of my colleagues first opened my eyes to using Docker and Makefiles together, I certainly wasn't aware of all the possibilities.

In this post, you'll be taken through some of the ways that Docker and GNU Make can effectively be used together. This will be shown by providing a simple Go example application, around which a Dockerfile and Makefile will be built. To follow along, you'll need to have at least a basic understanding of Makefiles and Docker.

Why Use Docker With Make?

Many developers already know why it makes sense to use Docker for your application. It helps you run things locally, ensuring that the environment is exactly what it will be when you run it on your servers. On top of that, it removes the need to install every tool locally, and instead allows you to simply run a `docker` command to have your application running.

Adding GNU Make to the recipe is where some people will fail to see the advantage. You'll hear some asking "Isn't Make an old tool?" or "Isn't it only meant for C and C++ projects?". In reality, this couldn't be farther from the truth. It's correct that Make is a utility developed back in the '70s and '80s, and yes, it's perceived as being tied to C and C++ applications, but that doesn't mean it doesn't have its advantages in other projects.

In the following sections of this chapter, you'll see just how useful Make can be when integrated into a Docker project. You'll see some of the simple advantages like not having to type out long commands, as well as some more advanced use cases like dynamically created Make targets for different Dockerfiles.

Integrating Make Into Your Docker Project

To see how you can integrate Make into your Docker projects, you'll first need to define a project to work on. As mentioned in the introduction, this tutorial will use a simple Go project for this. If you'd like to look at the completed project as a whole, the code for this tutorial can be found in this GitHub Repo.

Defining the Application First, you need to define the application itself. Create a new folder, and create a file called `main.go` inside of it. In `main.go`, paste the following code:

> The code for this specific step can be found in the branch "starter".

```go
package main

import "fmt"

func main() {
	fmt.Println("Hello World!")
}
```

If you're not familiar with Go, this is simply a "Hello World" example. You start by defining the package called `main`, after which you import `fmt`. `fmt` is to Go what "stdio" is to C++—it's the library used to communicate with the console. Next, the function `main` is defined, inside of which "Hello World!" is printed to the command line.

Go requires that a Module needs to be specified in order to build the application. This is done by simply running `go mod init` in your terminal. Once this is done, all that's left is to create the Dockerfile. Create a file named `Dockerfile`, and paste the following into it:

```dockerfile
FROM golang:1.18-alpine

WORKDIR /app

COPY go.mod ./
COPY *.go ./

RUN go build -o /hello-world

CMD ["/hello-world"]
```

If you've worked with Dockerfiles before, this should be very familiar. Start by defining the base image on line one, and then define the working directory inside the container. After that, you copy some files into the container, build the application by running `go build -o /hello-world`, and set the built binary to be executed when the container is spun up. You can make sure that everything works as expected by running `docker build --tag username/hello-world . && docker run username/hello-world`, replacing `username` with your own username in both instances. If the last line in your terminal is now "Hello World!", everything is working as intended!

Time to add a Makefile to the equation.

Building and Pushing the Application

The code for this step can be found in the "build-and-push" branch.

Start by creating a file called `Makefile`. In this file, you'll need to paste the following:

```makefile
DOCKER_USERNAME ?= username
APPLICATION_NAME ?= hello-world

build:
	docker build --tag ${DOCKER_USERNAME}/${APPLICATION_NAME} .
```

Again, be sure to replace the `DOCKER_USERNAME` variable with your own username. As you can see, this is a very simple Make target, and you can now run **`make build`** in your terminal to have your application built. At this point, you can start to see the advantages of using Make. You can now run just **`make build`** instead of **`docker build --tag username/hello-world .`**. Not only that, but if you integrate Make into all of your projects, you can always just run **`make build`**, and not even have to think about what the name of the project is.

The next step is to push the image. Again, this is a very simple target inside Make:

```
push:
	docker push ${DOCKER_USERNAME}/${APPLICATION_NAME}
```

You can now run **`make push`** to push the application to Docker Hub. If you're using your own Docker repository, remember to add that to the name of the image to make it work. You may still be unconvinced that it's worth using Docker and Makefile together. The really impressive part comes when you get used to not using those two make targets individually, but together:

```
$ make build push
```

This will become a huge time saver, especially if you integrate Make in all your Docker projects.

Releasing and Versioning the Application

The code for this step can be found in the "release-and-versioning" branch.

You've been shown how to build and push your Docker images with Make, but the biggest advantage when using Make is when it comes to releasing and versioning. With the way your Makefile is defined right now, all your images with be given the tag "latest". This isn't great, especially if you're using this in production and you accidentally execute **`make push`** locally.

Instead, let's add some functionality to the Makefile so it uses the Git SHA hash when building and pushing your image. Start by adding this variable to the top of your Makefile:

```
GIT_HASH ?= $(shell git log --format="%h" -n 1)
```

This gets the SHA hash from Git and stores it in the `GIT_HASH` variable. Now you can append this to both cases where you define the tag for your Docker image:

```
build:
	docker build --tag ${DOCKER_USERNAME}/${APPLICATION_NAME}:${GIT_HASH} .

push:
	docker push ${DOCKER_USERNAME}/${APPLICATION_NAME}:${GIT_HASH}
```

Now you can run **`make build push`** again, and see that it's now using the Git hash to tag your image. This is great for working on it locally, but what do you do when you want to actually push a **`:latest`** tag? In theory, you could overwrite the `GIT_HASH` variable when executing **`make build push`**, but that's more of a workaround. Instead, let's create a new target:

```
release:
	docker pull ${DOCKER_USERNAME}/${APPLICATION_NAME}:${GIT_HASH}
	docker tag  ${DOCKER_USERNAME}/${APPLICATION_NAME}:${GIT_HASH} ${DOCKER_USERNAME}/${APPL
	docker push ${DOCKER_USERNAME}/${APPLICATION_NAME}:latest
```

Three things are happening here. First, the image with the given Git hash is pulled from the Docker repository. This may seem excessive since the image has just been built, but is incredibly useful if, as an example, you're running this in a CI/CD pipeline where you don't want to build the application again, you just want to tag it with `latest` and release it. This is exactly what happens in the following two lines. The existing image is tagged with `latest`, and then it's pushed to the Docker repository.

You've now fully integrated Make into your Docker repository, and hopefully you can see the advantages that it'll bring to your workflow. This is all you really need to get started with using Make in Docker, but read on to see a more advanced use case.

Working With Multiple Dockerfiles

> The code for this step can be found in the branch `multiple-dockerfiles`.

There are instances where you want your project to contain multiple Dockerfiles. Maybe you have a repository that defines a bunch of different pipeline runners, or maybe you just want to have a version of your application that's easier to debug. Whatever the case, this step is useful if you have more than one Dockerfile.

As an example, let's take the case of wanting to have a version of your Docker image that has `make` installed, which is something our version of Alpine doesn't have by default. You don't want to add it to your production image, as it increases the image size, so instead you create a `Dockerfile.debug` where `make` is installed:

```
FROM golang:1.18-alpine

WORKDIR /app

COPY go.mod ./
COPY *.go ./

RUN go build -o /hello-world
RUN apk update
RUN apk add make # install make

CMD ["/hello-world"]
```

Now you have a debug version of your Dockerfile, but there's currently no way to build it using `make` commands. This requires some changes to your Makefile. First the existing targets have to be changed slightly:

```
_BUILD_ARGS_TAG ?= ${GIT_HASH}
_BUILD_ARGS_RELEASE_TAG ?= latest
_BUILD_ARGS_DOCKERFILE ?= Dockerfile

_builder:
	docker build --tag ${DOCKER_USERNAME}/${APPLICATION_NAME}:${_BUILD_ARGS_TAG} -f ${_BUILD_

_pusher:
```

```
	docker push ${DOCKER_USERNAME}/${APPLICATION_NAME}:${_BUILD_ARGS_TAG}

_releaser:
	docker pull ${DOCKER_USERNAME}/${APPLICATION_NAME}:${_BUILD_ARGS_TAG}
	docker tag  ${DOCKER_USERNAME}/${APPLICATION_NAME}:${_BUILD_ARGS_TAG} ${DOCKER_USERNAME}
	docker push ${DOCKER_USERNAME}/${APPLICATION_NAME}:${_BUILD_ARGS_RELEASE_TAG}
```

Two major changes have been made. First of all, the variables `_BUILD_ARGS_RELEASE_TAG`, `_BUILD_ARGS_TAG`, and `_BUILD_ARGS_DOCKERFILE` have been added. These follow the changes that have been made to the targets, which have been changed to `_builder`, `_pusher`, and `_releaser`, respectively. You can still use these targets as you have so far, like running `make _builder _releaser`, but if you're familiar with Make syntax, you'll know that the `_` at the start of these variables and targets indicates that they're not meant to be called from outside.

Instead, these targets are now internal. To restore the simple functionality of `make build push release`, we're going to create three new targets:

```
build:
	$(MAKE) _builder

push:
	$(MAKE) _pusher

release:
	$(MAKE) _releaser
```

Now you once again have the functionality of `make build push release`. This might seem redundant, which so far is correct. The exciting part about this is the next three targets that will be added to the Makefile:

```
build_%:
	$(MAKE) _builder \
		-e _BUILD_ARGS_TAG="$*-${GIT_HASH}" \
		-e _BUILD_ARGS_DOCKERFILE="Dockerfile.$*"

push_%:
	$(MAKE) _pusher \
		-e _BUILD_ARGS_TAG="$*-${GIT_HASH}"

release_%:
	$(MAKE) _releaser \
		-e _BUILD_ARGS_TAG="$*-${GIT_HASH}" \
		-e _BUILD_ARGS_RELEASE_TAG="$*-latest"
```

This is where the magic of this configuration really lies. To understand what's happening, take a look at the `build_%` target. It's using `%` and `$*` in combination to make for dynamic targets. In this case, if you execute `make build_debug`, it will build an image with the tag `debug-${GIT_HASH}` based on the `Dockerfile.debug` target. Now you can make endless variations of your Dockerfiles—without having to create multiple different make targets.

This does add some complexity to your project, but it makes everything much more dynamic and easier to work with in the long run.

Conclusion

By now, you've seen how you can easily and quickly add Make to your project. The advantages of using Make can range from simple use cases like avoiding typing out long commands, getting everyone on the team used to the same syntax in all projects, and even being able to create dynamic targets that create new possibilities for you and your team.

How To Use Makefiles on Windows

As the field of DevOps and build release engineering continues to grow, many new tools are being developed to help make building and releasing applications easier. One of the tools that has been in use for many years is Make, which is still heavily used by engineers today.

A *Makefile* is a simple text file consisting of targets, which can invoke different actions depending on what has been configured. For example, with a Makefile, you can invoke a build of your application, deploy it, or run automated tests and it can dramatically increase the efficiency of your workflow.

Initially, it was Stuart Feldman who began working on the Make utility back in 1976 at Bell Labs. However, the version of Make most commonly used today is GNU Make, which was introduced in the late 1980s.

While the tool was originally meant to run on Linux, Make's popularity has interested those working on other operating systems as well. There are several ways to run Makefiles on Windows, and in this chapter you'll be introduced to each option and learn about their strengths and weaknesses.

Using Make on Windows

windows

Before looking at the different options available, you should know why you want to run Makefiles on Windows in the first place. Or rather, if you're working on Windows, why are you even interested in Makefiles?

Historically, the biggest reason for wanting Makefiles to run on Windows is that the developers in your organization are working on Windows. Seeing as how the de facto standard for languages like C and C++ is to use Make, it's no wonder that Windows users want the ability to use Make as well.

As applications and infrastructure become more modern, the cloud is another reason for wanting Makefiles on Windows. Many infrastructure engineers want their applications to be run on Linux, likely led by the adoption of tools like Docker and containerization in general. Additionally, on Linux, a Makefile is the primary tool to use in many cases, especially when it comes to building native Linux applications. However, many engineers are still using Windows on their workstations, leading to the question of how to run Makefiles on Windows. Let's dive into the possible answers.

Chocolatey

chocolatey

Linux users have been using package managers for decades, yet they've never gained much traction on Windows. Up until the release of winget, the concept of a package manager was never something that was natively included on Windows. Instead, Rob Reynolds started working on an independent package manager back in 2011 that would come to be known as Chocolatey.

Chocolatey is now widely used on Windows to install packages, and you can use it to install `make` as well.

To do so, run the following command in an Administrative PowerShell window:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProto
```

You can find the newest installation instructions at any time on the Chocolatey website.

Once Chocolatey is installed, you may have to close down the PowerShell window and open it back up. After that, run the following command:

```
choco install make
```

Once the script is done running, `make` will be installed. You may need to restart the PowerShell window again, but at this point you are ready to use Makefiles on Windows.

Chocolatey will likely be the most popular option for those who want to stick to a pure Windows installation. It's easy to install, easy to use, and you don't need to jump through any hoops or workarounds to get it working.

At this point, you can use `make` just like you otherwise would, and you can test it by running `make -v`.

Cygwin

Historically, one of the most popular ways of running any type of Linux functionality on Windows has been to use Cygwin. Cygwin aims to give a Linux feeling to Windows by holding a large collection of GNU and open source tools. It's important to note that this does *not* mean it will give you native Linux functionality. However, it does allow you to use Linux tools on Windows. There's a big difference between the two; for instance, Cygwin does not have access to Unix functionality like signals, PTYs, and so on. It's a great tool for when you want to use familiar Linux commands but still want them to be run on Windows.

To use Cygwin for Makefiles, start by downloading and installing Cygwin. During the installation, you'll see a window popping up asking you what packages you want to install. In the top left corner, make sure to select **Full** and then search for `make`.

Searching for "make"

Figure 12: Searching for "make"

Your search will give you a list of several different packages. You want to choose the one that's labeled just as `make`. Change the dropdown menu where it says **Skip** to the latest version.

Choosing "make"

Figure 13: Choosing "make"

Now you can finish the installation by clicking **Next** in the bottom right corner. Once the installation is done, you can open up Cygwin and verify that `make` has been installed by executing `make --version`.

NMAKE

One of the alternatives that you'll often hear about regarding running Makefiles on Windows is NMAKE. While it is an alternative to `make`, note that you cannot simply take your existing Makefiles from Linux and run them using NMAKE; they have to be ported.

First of all, the compilers are different on Windows and Linux, so if you are specifying your compiler in your Makefile, you'll have to change that to whatever is relevant on Windows. At the same time, you'll have to change the flags that you send to the compiler, because Windows typically denotes the flags using `/` instead of `-`.

On top of that, it doesn't recognize all the syntax that you're used to from GNU Make, like `.PHONY`. Lastly, Windows obviously doesn't recognize the commands that work on Linux, so if you have specified any Linux-specific commands in your Makefiles, you'll also have to port them.

All in all, if your entire organization uses Windows and you simply want the typical functionality of GNU Make, then NMAKE is a viable solution. However, if you just want to quickly run your traditional Makefiles on Windows, NMAKE is not the answer.

CMake

cmake

As with NMAKE, CMake is not a direct way to run your Makefiles on Windows. Instead, CMake is a tool to generate Makefiles, at least on Linux. It works by defining a `CMakeLists.txt` file in the root directory of your application. Once you execute `cmake`, it generates the files you need to build your application, no matter what operating system you're on.

On Linux, this means that it creates Makefiles for you to run, but on Windows it may mean that it creates a Visual Studio solution.

CMake is a great solution if you don't care too much about running Makefiles specifically, but you want the functionality, namely the ease of use in a build process, that you can get from Makefiles.

Windows Subsystem for Linux

The Windows Subsystem for Linux (WSL) is an honorable mention. It's cheating a bit to say that it's a way to run Makefiles "on Windows," as your Makefiles won't actually be running on Windows.

If you haven't heard of WSL before, here's an extremely oversimplified explanation: It uses Hyper-V to create a hyper-optimized virtual machine on your computer, in which it runs Linux. Basically, you get a native Linux kernel running on your Windows computer, with a terminal that feels as if it's part of Windows.

You should look into WSL if what you care about most is having Windows as your regular desktop environment, but you're fine with all of your programming and development going on inside of Linux.

Conclusion

As you can see, there are a few different ways you can be successful in running Makefiles on Windows. However, you do need to be wary of the fact that it will never be a perfect solution. Every solution is in some way a workaround, and the closest you'll get to feeling like you're using native Makefiles while using Windows is to install something like WSL.

Getting a Repeatable Build, Every Time

Repeatability Matters

In our journey to becoming better software engineers we have learned of various ways in which the team's productivity could be improved. We noticed that a focus on build repeatability and maintainability goes a long way towards keeping the team focused on what really matters: delivering great software. Many of these ideas helped shape what Earthly is today. In fact, the complexity of the matter is what got us to start Earthly in the first place.

I wanted to sit down and write about all the tricks we learned and that we used every day to help make builds more manageable in the absence of Earthly. It's kinda like a "what would you do if you couldn't use Earthly" chapter. This will hopefully give you some ideas on best practices around build script maintenance, or it might help you decide on whether Earthly is something for you (or if the alternative is preferable).

In this chapter, we will walk through the 10,000 feet view of your build strategy, then dive into some specific tricks, tools, and techniques you might use to keep your builds effective and reproducible, with other off-the-shelf tools.

Putting Together Complex Builds - Assumptions

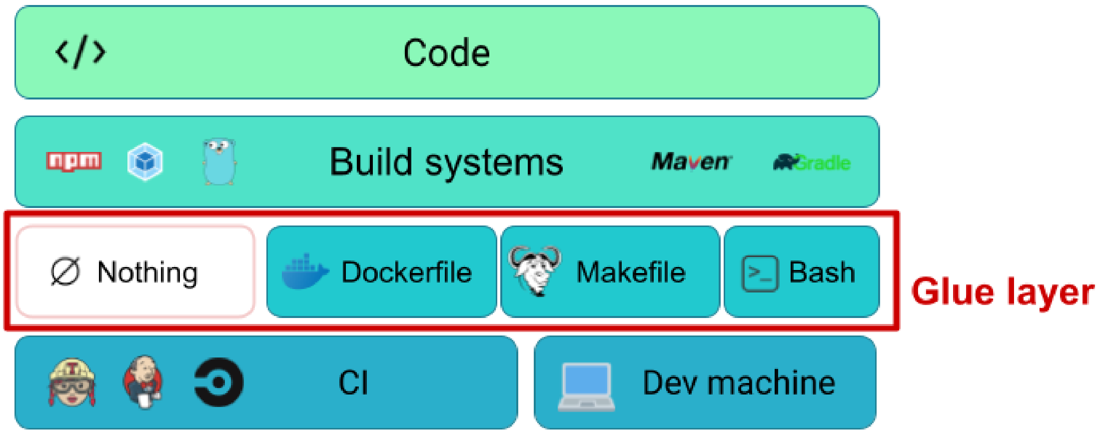
Since this is a guide about how not to use Earthly, we'll try to achieve the same benefits of Earthly, using other tools. Here are some assumptions:

- You are building cloud or web services, targeting Linux.
- You have a setup where multiple programming languages come together (e.g. multiple microservices).
- You would like to have fast local dev-test cycles for individual components.
- You would like to be able to iterate quickly when there is a CI failure.
- Developers on your team use varying platforms for day-to-day development: some Linux, some Mac, some Windows.
- You would like to optimize for cross-team collaboration.

Scope: The Glue Layer

We will focus primarily on the glue layer of your builds. The stuff that brings everything together - maybe it packages things up for releases, or maybe it prepares packages for deployment, or perhaps it is simply a script that the CI definition calls into.

This is a diagram we sometimes use to describe the glue layer. In this chapter, we'll be focusing on the Dockerfile, Makefile, and Bash parts of that glue layer. Not having a glue layer can make CI failures difficult to reproduce, or for other teams unfamiliar with the language-specific build tooling to effectively create the right environment to run builds.



The glue layer is the layer between the various projects that need to be built and will act as the common denominator - a vendor-neutral build specification. If we don't choose such a glue layer, then the CI YAML (or Groovy?) becomes the glue layer and that would mean that it's more difficult to run it locally for fast iteration.

Because we want to encourage cross-team collaboration, we want to standardize the tooling across teams as much as possible. Different language ecosystems will have different tools and we want to keep using those. You can't tell the frontend team not to use package.json / npm / Yarn etc - that would be terribly cumbersome for them. So we're not touching the language-specific build layer.

In addition, because we want to be able to iterate quickly when there are CI failures, we will containerize the build as much as possible. If the failure is part of a containerized script, then it will be easier to reproduce it locally.

A popular choice for the glue layer is to use Makefile + Dockerfile. Makefiles are great as a collection of everyday commands that we can use (`make build`, `make start`, `make package`, `make test`), while Dockerfiles help keep most of the build containerized.

Another option might be to use bash + Dockerfile, where a collection scripts for everyday tasks exist under a `hack` or `scripts` directory (`./hack/build`, `./hack/test`, `./hack/release`).

Yet another, more exotic, option is to use another scripting language, such as Python or JavaScript (see, for example, the zx library). Keep in mind, however, that it's best when most of the engineers can read and write the build scripts with ease. Sometimes too much flexibility of the programming language can make it harder for others to read and understand the scripts. This is an especially important point as the build scripts will likely be read much more often than they will be written.

For this guide, we'll use Makefile + Dockerfile, as an arbitrary choice. Note, however, that neither Bash nor Makefile is very intuitive at first glance and you will need to help out junior developers, or developers that simply haven't had the opportunity to learn these yet. And also, these are purely arbitrary choices, based on what we see as popular technologies used in this area. Your own choice may vary for good reasons. This guide is not saying that Makefile / Bash / Dockerfile are the only right choices.

Tips for Taming Makefiles in Large Teams

While there is ShellCheck for linting shell scripts and avoiding the typical pitfalls, there is currently no linter for Makefiles. Part of the reason is that there are multiple styles or philosophies as to how to write Makefiles that it is hard to enforce specific rules that prevent human errors. For example, take a look at this particularly opinionated approach.

Makefiles have the ability to manage the chain of dependencies and avoid duplicate work. However, this feature is heavily based on file creation time (if an input is newer than the output, then rebuild the target), and in most cases that is not enough. If you really want to, it is possible to force everything into that model. However, that tends to create very complicated Makefiles that often only one person on the team understands. This person becomes the "Build Guru" and all build maintenance works flow through them. This "Build Guru" dynamic is very common but best avoided. With Makefiles, using a limited subset of the Makefile language can help avoid this pattern and keep everything more maintainable for the entire team.

If the Makefile wraps an existing build (for example, a Gradle-based or npm-based build), it might make more sense to avoid Makefile's more advanced features in those particular projects, to help the team be comfortable with making changes to the Makefile, with limited knowledge. In such cases, the caching would be handled by the wrapped build system and not by the Makefile itself.

Here are some guidelines to help keep Makefiles simple and understandable, when the Makefile wraps another build system.

- Only use `.PHONY` targets. Explanation: regular Makefile targets are names of files that need to be output. If the file does not exist or it is older than its inputs, `make` decides to run the target. A `.PHONY` target is a target that is not really a file - it's just a name that the user can use as a short-hand to refer to a recipe to be run.`.PHONY` targets don't use the freshness algorithm.
- Avoid overusing the distinct features of the different variable flavors (`=` vs `:=` vs `?=` vs `+=` vs `!=`). Someone not knowing the details of how Makefiles work will be confused if the logic relies heavily on the specific flavor.
- Avoid order-only prerequisite types (`targets : normal-prerequisites | order-only-prerequisites`). The `order-only-prerequisite` can be misleading to someone new to Makefiles.
- Avoid complex rules involving `patsubst` or special variables like `$<`, `$^`, `$*`, `$@`. The way these work is a mystery to Makefile newcomers and it can be really hard to Google for the meaning.

In other words, try not to be too smart about Makefiles. Note, however, that if the Makefile does not wrap an existing build system, then it is likely that you'll need the advanced features of Makefile to correctly make use of the caching system.

Another thing to consider is that certain UNIX commands vary from platform to platform. For example, `sed` and `find` have important differences between GNU/Linux and macOS, and while Windows can behave very much like Linux through WSL 2, testing is needed to verify everything you are using will work the same. Ask a colleague to test out your scripts on their platform if in doubt.

Remember the Tab

Use of the tab character is mandatory in Makefiles or you'll get this error. This usually catches newbies off-guard.

```
Makefile:273: *** missing separator. Stop.
```

Tips for Dockerfiles

To help keep builds reproducible, it's useful to have a version of the build in containerized form. If performance allows, this might be the day-to-day form that everyone uses. If not, at the very least, it's the form that is used to debug CI failures.

Dockerfiles are the bread and butter of containerized builds. However, they have an important limitation: they can only output Docker images. They weren't designed to run unit tests or to output regular artifacts (binaries, jars, packages, etc). However, with some wrapper code, it is possible to extract regular files from images or to execute unit tests too. The following will illustrate how.

Use Multi-stage Dockerfiles or Multiple Dockerfiles

To help split up Dockerfile recipes into logical groupings, you can make use of multiple Dockerfile targets (also called multi-stage builds) and/or of multiple Dockerfile files.

To define targets use `FROM ... AS ...`

```
FROM alpine:latest AS my-target-name

...

FROM ubuntu:latest AS another-target-name

...

FROM my-target-name AS yet-another-target-name

...
```

A target can be used in another `FROM` command or a `COPY` command (`COPY --from=my-target-name <file-from> <file-to>`) to copy resulting files from one target into another. This flexibility allows your definition to be more modular, and possibly also more efficient in cache, or more efficient in the size of the final image.

To invoke a specific Dockerfile target, you can use the `--target` flag of `docker build`.

```
docker build --target=my-target-name -t my-build-image .
```

If a target is not specified, `docker build` will simply build the last target in the file.

Running Containerized Unit Tests

In order to run unit tests in containers, you might take either of the following strategies:

Run the test as a Dockerfile target

Build a container with all the necessary code and set the entrypoint to execute the unit tests, then run that container.

A third, less recommended, option is to execute tests as additional layers of the final image. This is not a great option because it unnecessarily bloats your image size and also, there's no way to skip them. If you just want an image quickly this option will end up getting in your way.

Option a. has the advantage that if nothing changes, the Dockerfile caching will simply reuse previous work and return quickly. Since you don't need the image being created, you can simply skip mentioning a tag via `-t`:

```
docker build .
```

Option b. requires managing an extra image, but might make them look more like the integration tests. Plus, if you want to mount in the source code instead of `COPY`ing it (faster on Linux), this option allows that.

Running Integration Tests

When running containerized integration tests that depend on additional services, only **option b.** above is available. In such cases, you might want to make use of docker-compose to manage multiple containers running together. Even if your setup is small, it's just easier to be able to kill everything at once if your tests hang.

Another option is to use testcontainers - a language-specific library that allows you to bring up containerized helper services during integration testing. This can be a great abstraction, however, if you want to keep the integration tests containerized too, you'll need to mount `/var/run/docker.sock` (this is sometimes referred to as "docker-out-of-docker"). This will allow the integration test code to bring up the necessary helper services via testcontainers.

```
docker run -v /var/run/docker.sock:/var/run/docker.sock ...
```

Either way, to keep your test suite as easy to use as possible, it's great if you can map the series of commands needed to run the integration tests to a single make command (e.g. `make integration`) and use this command without any additional settings or wrappers in your CI scripts too. That way, if your CI breaks, then you have a very easy one-liner that should (hopefully) reproduce the failure on your local machine.

Outputting Regular Files

Even the most containerized builds benefit from being able to occasionally output regular files. Here are some possible situations:

- Binaries / Packages / Library archives
- Releasables (deb packages, source code tarballs)
- Screenshots from UI tests
- Test coverage reports
- Performance profile reports
- Database schema init scripts

- Generated source code files (to help IDEs with syntax highlighting)
- Generated configuration

To output regular files as part of a containerized build, there are a few options.

Generate the file(s) into a host-mounted volume as part of `docker run`.

Generate the file(s) as part of `docker run` and then extract them using `docker cp`.

Generate the file(s) as the contents of an image during `docker build` then extract them using the `docker build -o`option.

Generate the file(s) as the contents of an image during `docker build` then extract them using `docker cp`.

Let's take these one at a time:

Option a. Generate the file(s) into a host-mounted volume as part of `docker run`: This is by far the most used, but arguably the least useful. This technique involves setting the entrypoint as the command that generates the resulting artifact and mounting an output directory (or maybe even the entire source code directory) where the result can be stored and shared with the host system.

```
docker run --rm -v "$PWD/output:/output" my-image:latest
ls /output
```

The typical issue with this approach is that the resulting artifacts are owned by `root`. Getting rid of them or moving them around then requires `sudo`. There are, of course, ways to generate the files as a different user within the container, but this can be cumbersome.

Option b. Generate the file(s) as part of `docker run` and then extract them using `docker cp`: This option involves executing the `docker run` and then afterward issuing `docker cp` to extract the resulting artifact. Here's an example:

```
docker run --name build-artifact my-image:latest
docker cp build-artifact:/output/my-artifact ./output/my-artifact
docker rm build-artifact
```

This option will produce results owned by the right user. The one possible downside of this approach is that if the `docker run` fails, a hanging `build-artifact` container remains, which will cause the next run to fail due to naming conflict. This can be easily fixed, however, by adding `docker rm -f build-artifact` at the beginning of the script.

Option c. Generate the file(s) as the contents of an image during `docker build` then extract them using the `docker build -o` option: This is a lesser-known technique particularly suited for outputting entire directories. It essentially outputs the root directory of an entire image to the host. Here's how this might be used.

```
FROM alpine AS base
RUN mkdir -p /output
RUN echo "contents" >/output/my-artifact
```

```
FROM scratch
COPY --from=base /output/* /
```

```
docker build -o ./output .
cat ./output/my-artifact # prints "contents"
```

The use of the scratch base image is such that the final image is minimal and only the required artifact is copied over to the host.

This technique outputs files owned by the right user, however sometimes the fact that it outputs a whole directory can be limiting.

Option d. Generate the file(s) as the contents of an image during `docker build` then extract them using `docker cp`: This technique is somewhat similar to **option b.** except that no `docker run` is used.

```
docker build -t my-image:latest .
docker create --name output-artifact my-image:latest
docker cp output-artifact:/output/my-artifact ./output/my-artifact
docker rm output-artifact
```

This option too will produce artifacts owned by the right user.

Tips for Taming Bash Scripts

Regardless of the build setup, bash scripts are sometimes inevitable, especially when extensive release logic is needed and the language-specific tooling doesn't offer specific support.

If you're new to bash scripting, I'll run through a few of the more useful beginner tricks from our understanding bash article.

- Spaces are surprisingly important. `A=B` is different from `A = B`. Also `if[$A=$B]` does not work - it has to be `if [ $A = $B ]`.
- Variables can cause really weird things if they are not wrapped in double-quotes. The reason is that a variable that contains spaces can expand across multiple command parameters unless it's within quotes. To prevent any surprises, a good rule is to never expand a variable outside of double-quotes. So use `"$ABC"` always and not simply `$ABC`.
- A really good tool to check for some common errors like the above is `shellcheck`.
- If there is an error in a bash script, by default, the script simply continues without a worry. This is very often not what you want - you want to terminate immediately so that the user is aware that something has failed. To enable this, put `set -e` at the top of the file.
- A series of piped commands can also fail and bash doesn't terminate by default. To terminate immediately in such cases, you can enable `set -o pipefail`.
- An undefined variable is treated as the empty string in bash. However a lot of times an undefined variable can also be caused by a typo, or an incomplete rename during a refactor, or some other human error. To avoid surprises, you can do `set -u`, which will cause the script to fail on undefined variables. To initialize a variable with a default value (and avoid this error, if you'd like to allow an optional external variable), you can use `: "${<variable-name>:=<default-value>}"`. Just don't forget the `:` in front of it.
- If you have no idea what your bash script is doing, you can make it print every statement being evaluated using `set -x`. It's especially useful when debugging.

Importing Code and Artifacts from Another Repository

Sometimes it's useful to import the result of one build from repo A into the build of repo B. Here are some examples of situations where this is needed:

- Building some binaries in a core repo and packaging the release in a "build-tooling" repo (example in Kong)
- Generating protobuf files in one repo and reusing those files in multiple other repos (client(s) and server)
- Pre-computing some initialization data in one repo and using it in another repo
- Cases where the language-specific tooling offers little or no support for importing code from another repository

There are multiple ways to achieve importing on your own. In general, if the language you use provides solid importing mechanisms, there is little to no reason to build any scripting that does the same thing. In other cases, you might go with one of the following makeshift options:

Git clone

Submodule

Pass files via a Docker image

Pass files via S3

Option a. Git clone: This relies on git-cloning one repo from within the other. It's especially useful when the code itself is needed and not necessarily an artifact resulting from building that code. Here how this might be achieved in a Makefile:

```
my-dep: ./deps/my-dep

./deps/my-dep:
  mkdir -p deps
  rm -rf ./deps/my-dep
  git clone --branch v1.2.3 <clone-url> ./deps/my-dep

clean:
  rm -rf ./deps
```

Although it's relatively crude in that it wipes the whole dir and re-clones it when necessary, it's also pretty robust because there's nothing assumed about the state of that directory. (Just make sure you're not making important changes in that directory as those may be lost!)

The `--branch` setting takes any git tag, sha, or branch, or, in general, a git ref. This helps pin the dependency to something specific if that is needed.

You'll need to be careful about how you use the `<clone-url>` as some engineers will use HTTPS auth, while others will use `ssh` auth. You might want to standardize on only one of the two options and use git's `insteadof` feature if an individual developer would like to switch to the other URL.

For example, if a developer wants to dynamically switch from `https://` to SSH GitHub URLs, they can do:

```
git config --global url."git@github.com:".insteadOf "https://github.com/"
```

Option b. Submodule: This is another popular option especially when the source code itself is also needed. The technique involves relying on the `git submodule` capabilities built into git.

The way it works is that git marks certain paths within the repository where submodules (other git repositories) can be cloned into. You can use the following command to add such submodules:

```
git submodule add <clone-url> <path>
```

This command clones the referenced repository into that path and also adds a `.gitmodules` file in the root of your repository, which might look like this.

```
[submodule "<name>"]
        path = <path>
        url = <clone-url>
```

Besides this entry, git internally also stores the git sha of the submoduled repository, so it is tied strongly to a specific version of it.

Although there is significant tooling out there for working with submodules, some people don't enjoy them because they require adjusting the typical git workflow to account for maintaining the submodules. In some cases, when submodules are overused, they can create more confusion than necessary. You can tell things can get complex when git comes with commands like `git submodule foreach 'git stash'`.

There is much more to submodules than is in scope for this guide, but a good run-down can be found in the git official documentation.

Option c. Pass files via a Docker image: This option involves creating a Docker image that is not really actually run. It is merely used as a package repository.

The advantage of this technique is that Docker images support tags, which allow for embedding into a release-process-based workflow. This technique is especially useful when artifacts beyond the source code itself are needed. Things like binaries, or generated files.

However, this option does require repo A to execute a build that packages up the image before it can be used in repo B. If **options a.** and **b.** are simply **commit to repo A -> use in repo B**, for **option c.** the sequence is **commit to repo A -> wait for CI build of repo A to complete -> use in repo B**.

Side Note. If the CI is slow, this can be a productivity hog. To counter this situation, make sure that the individual engineer on the team can build the image independently from the CI, in order to be able to iterate locally quickly.

Here's how this option works in general.

Repo A:

```
FROM scratch
COPY ./build/files ./
```

```
docker build -t my-registry/my-artifacts:my_tag .
docker push my-registry/my-artifacts:my_tag
```

Repo B:

```
docker pull my-registry/my-artifacts:my_tag
docker create --name output-artifacts my-registry/my-artifacts:my_tag
docker cp output-artifacts:files ./deps/files
docker rm output-artifacts
```

The nice thing about this approach is that Docker will use its cache if an artifact has not changed.

Option d. Pass files via S3: This option is very similar to **option c.**, except that it uses an S3 bucket instead of a Docker registry for storing artifacts.

Although I've heard this being used as a viable alternative, I've never seen this in action myself. With the wide range of Docker registries available out there, I don't think this option is necessarily better. If, however, your company for whatever reason cannot provide you with a Docker registry repository, just know that using S3 (or any other cloud blob store) is also a possibility.

Parallelism With Makefiles and Dockerfiles

The bread and butter of improving build speed are caching and parallelism. We will look at each of these topics in the context of Makefiles and Dockerfiles.

Makefiles have parallelism support out of the box. Running any make target with the `-j` option tells Make to execute dependencies in parallel (e.g. `make -j build`). Although this is a seemingly easy win for performance, it comes with a number of strings attached, that you should be aware of.

First of all, Make targets are not isolated - they all operate on the same directory. When you run multiple such targets in parallel, you may get very surprising results due to race conditions. Imagine if, for one of your targets, you really want to run a clean build, so you make it depend on the target `clean`. And then you have another target, which ensures that your `build` directory has been created. Well if these run in parallel, you can imagine how they could get in each other's way.

To really make use of this option, you'll need to design target dependencies with parallelism mindfulness. You'll need to really think about dependencies whether they need to be built in a certain order, or if they can be built in any order.

If you need dependencies to be built in a specific order, a great way to achieve that is to use recursive Make calls:

```
build:
  $(MAKE) dep1
  $(MAKE) dep2
  $(MAKE) dep3
  actually build
```

If dependencies can execute in any order, then declaring them as regular target dependencies will be fine:

```
build: dep1 dep2 dep3
  actually build
```

Note that these principles may be difficult to uphold in a large team, as not all engineers will be running the build with the parallel option turned on. So a lot of times the parallelism capability is not tested, and as a result, it can break often. Keep these things in mind as you may need to reinforce the use of parallelism in project READMEs to hopefully be better supported by individual PRs.

Another, either alternative or complementary option, is to make use of Dockerfile parallelism. With BuildKit turned on (`DOCKER_BUILDKIT=1 docker build ...`), Dockerfiles are built with parallelism automatically, if they involve multiple targets. So for example, if you have targets `dep1`, `dep2` and `dep3` and you `COPY` files from them like so:

```
FROM alpine AS build
COPY --from=dep1 /output/artifact1 ./
COPY --from=dep2 /output/artifact2 ./
COPY --from=dep3 /output/artifact3 ./
RUN actually build
```

Then `dep1`, `dep2`, and `dep3` will be executed in parallel. The nice thing about this is that there is nothing shared between these three targets (unlike the Makefile case) and they will just work in parallel. No special considerations are necessary.

Shared Caching

Another way to speed up builds is to make use of shared caching. Shared caching is especially relevant in modern CI setups where the CI task is sandboxed. Because the CI is sandboxed, it cannot reuse the result of a previous build.

There are, of course, CI-supported cache saving features. However, those features are not usable when testing locally. And in the end, all that CI caching does is just upload a file or directory to cloud storage, which can be downloaded later in another instance of the build.

Dockerfile builds have pretty powerful cache saving and importing capabilities, thanks to the new BuildKit engine. To use these capabilities, you need to first enable BuildKit (`DOCKER_BUILDKIT=1 docker build ...`) and then pass the right arguments to turn these features on.

First, you need to push images together with the cache manifest:

```
DOCKER_BUILDKIT=1 docker build -t my-registry/my-image --build-arg BUILDKIT_INLINE_CACHE=1 .
docker push my-registry/my-image
```

And then you need to make use of the cache in subsequent builds using `--cache-from`:

```
DOCKER_BUILDKIT=1 docker build --cache-from=my-registry/my-image -t my-registry/my-image .
```

Doing both in the same command looks like this:

```
DOCKER_BUILDKIT=1 docker build --cache-from=my-registry/my-image -t my-registry/my-image --build-
docker push my-registry/my-image
```

If the image does not already exist (or if it does not have a cache manifest embedded), there will be a warning, but the build will work fine otherwise.

Managing Secrets

Mature CI/CD setups often hold more secrets than there are available in the production environment. That's because they often need access to registries, artifact repositories, git repositories, as well as staging environments where additional testing can be performed, and also the production environment itself, where live releases are deployed to, together with schema write access to DBs for upgrades, and also possibly S3 access. Wow - that's a mouthful.

Oh, did I also mention that the CI runs a ton of code imported from the web? That `npm install` one of your colleagues ran in CI will download half the internet. And not all of it is particularly trust-worthy.

This whole thing makes for an incredibly risky attack surface. Giving access to build secrets to every developer adds an unnecessary extra dimension to the whole risk.

However, to be able to reproduce some of the builds, you really need to give at least *some* access. At the very least, developers need read access to most Docker registries and artifact repositories / package repositories. In fact, if you want to ensure that artifacts and images that make it to production are never created on a developer's computer (because you don't know if their specific environment will produce the releasable correctly consistently), you can simply not give out write access to any Docker repository and make it a rule that only the CI may have write access. (You might still need separate, non-production repos for local testing, though).

For the read access, it's best if each engineer gets their own account and credentials. In case someone is terminated, you can simply revoke access from that engineer's account and know that all other credentials are not compromised.

For reproducing CI/CD failures in pipelines with more sensitive access, you will need to maintain a small list of employees who will receive more privileged production access.

To share the initial credentials with each engineer, if an email invite feature is not provided by the system, it's best to use a password manager to send credentials one-to-one. Sometimes this process gets fairly manual in larger teams, but for the reasons mentioned above, it's important to maintain segregated access.

For a seamless setup, you can also store build credentials in HashiCorp Vault and use the Vault CLI in scripts to read secrets where they are needed.

```
vault read -field=foo secret/my-secret
```

This may be helpful in setups where extensive use of build credentials is required. Managing a Vault installation, however, is a whole project of its own, and going into those details is perhaps a story for another day.

Maintaining Consistency

Now to turn to a more philosophical aspect of builds. A great build setup is one where every engineer can understand what it does fairly quickly and where every engineer can contribute with minimal to no help at all. Although these principles are obvious goals to have, the norm is quite the opposite.

Engineers don't have an innate need to mess things up. It's not like we wake up in the morning trying to make builds cumbersome and complex for everyone else. The issue is usually that builds simply grow organically into an eventual big hairball of mess.

The result of this is that new team members have difficulty making sense of the build scripts and colleagues from other teams have difficulty in contributing to your team's project, thus creating integration gaps.

Think about how this is in your organization. Is it difficult for you to contribute to another team's codebase? Do you know how to run the project's unit and integration test suites? Is there even an integration test between the two projects?

To help teams meld with each other, it's particularly important to have a standard way to build and test any project in the organization. If the build is containerized and everyone has read access to the right base images, then everyone can build and test each other's codebases, thus eliminating at least one of the natural barriers that get formed between engineering teams.

If you want to go a step further, you can even standardize a set of commands for the repositories across the organization. For example:

- `make build` - builds the project
- `make test` - runs the unit tests
- `make integration` - runs the integration tests
- `make ci` - executes the same script that is run in CI

Then your engineers will be able to build another team's codebase in their sleep!

Other Options

This book wouldn't be complete without mentioning that if you're a large organization (thousands of engineers), you can also take a look at some large-scale build systems as popular alternatives to in-house scripts. The names that come to mind are Bazel, Buck and Pants. These systems imply heavy organization buy-in and well-staffed build engineering teams to manage them.

Some of these can provide some of the most advanced capabilities available on planet Earth, however, they do come with the tradeoff that they don't integrate well with most open-source tooling and so all projects need to be adapted to fit the paradigm.

If you'd like to get started exploring these, we have previously written about Monorepos and Bazel.

Parting Words and Mandatory Plug

Getting everyone to write containerized builds is difficult in a growing organization. As you can see from this book, certain operations within containerized builds are not trivial to achieve and the wheel may be reinvented many times across the different teams.

For these reasons, we have built Earthly. Through Earthly, we wanted to give containerized builds to the world, for the sake of reproducibility. From our own experience, we saw that Dockerfiles alone are not meant as build scripts, but rather as container image definitions. In true Unix philosophy, they are a great tool for that specific job - they do one thing and they do it well. To go the extra step and have containerized builds scripts (not just image definitions), a number of tricks and wrappers are necessary. Earthly takes the best ideas from Dockerfiles and Makefiles and puts them into a unified syntax that anyone can understand at-a-glance.

Give Earthly a try and tell us what you think via our Slack or our GitHub issue tracker.