Build Tools (like a compiler)

Audiences for build tools:

- **Coders:** These people write the actual source code.
- **Builders:** These people are in charge of assembling the pieces of the program into a coherent application. They "run make".
- **Distributors:** These people take the built products and ship them out to the users. They "run Debian".
- **Installers:** These people take the output of the distributors and install them on the machines.
- **Testers:** These people try to find problems in the code or any of the steps after it like distribution and installation. These are often done by people lower in the chain, like interns.
- **Users:** These people are the intended consumers of the product.

No Build Tools?

You can write your own project-specific build tools. Suppose you have a simple example of multiple C files that you need to link together. You could write a build script `build.sh`:

```
$ gcc -c x.c
$ gcc -c y.c
$ gcc -c z.c
$ gcc -o foo x.o y.o z.o
```

> This is a simple build script for compiling and linking multiple C files together into a single executable using GCC (the GNU Compiler Collection).
>
> `$ gcc -c x.c`
> > `-c` tells GCC to compile the source file `x.c` into a corresponding object file named `x.o` but not to link it.
> The object file contains the compiled code of your source file in a format that can be linked with other object files to create an **executable**.
>
> `$ gcc -o foo x.o y.o z.o`
> This final line links the three object files `*.o` together into a single executable file called `foo`.
> > `-o foo` specifies the name of the **executable** file.

> Your goal as a software developer is to automate yourself out of a job. You constantly want to take the things that you're doing and automate it. **- Dr. Eggert, probably**

Make and Makefiless

`Make` is a build automation tool that automatically builds executable programs and libraries from source code.

`Make` was inefficient by default because it would compile the entire program even if only one file was edited, so it does a lot of unnecessary work.

> However, the existence of `Makefile` improve `make` efficiency in managing project dependencies and minimizing build times. It achieves this by only recompiling parts of a project that have changed since the last build, rather than recompiling everything.

A `Makefile` is a configuration file that contains a set of directives used by the make build automation tool. Thus, the essence of the **Makefile** is that you write a set of **recipes** in it, each of which instructing how to build a file. When you run `make`, the program does the *minimum amount of work* necessary to build the file. It does this with the concept of **dependencies**.

> **Q: Relationship between Make and Makefile?**
>
> A: `make` is the program that automates the build process, and the `Makefile` is the blueprint it follows to do so. The `Makefile` defines what `make` needs to do, making the build process efficient by only updating what has changed since the last build.

Basic Structure of a Makefile

`Make` is like a hybrid language. The `target: dependencies` lines are its own `Makefile` language. The recipes are in the shell language, commands that you would write at the terminal.

The basic structure of a `Makefile` rule is:

```
target / FILE_OUTPUT: dependencies
    recipes
```

target / FILE_OUTPUT: These are usually the names of files to be generated, such as executables or object files. A target can also be **a label** for a group of commands that **NOT** produce a file.

dependencies: These are files that the target depends on to be created or updated. If any dependency has changed since the last build, the target will be rebuilt.

recipes: These are the shell commands executed to build a target. Commands must be preceded by a tab character.

For example:

```
CC = gcc
x.o: x.c
  $(CC) -c x.c
y.o: y.c
  $(CC) -c y.c
z.o: z.c
  $(CC) -c z.c
foo: x.o y.o z.o
  $(CC) $< -o $@
```

```
    # Escaping the $ so the shell can see it
    echo $$PATH
```

Automatic Variables

Makefiles support special variables within a rule:

`$@` represents the target

`$^` represents all the dependencies

`$<` represents the first one dependency

For example:

```
output: input1.o input2.o
    gcc -o $@ $^
```

which can be expanded to:

```
$ gcc -o output input1.o input2.o`
```

Ignoring Errors

`-` (dash) instructs `make` to ignore errors from the command.

In a `Makefile`, prefixing a command with a `-` (dash) instructs make to ignore errors from that command. This feature is particularly useful for `clean-up` targets that might attempt to delete files which may not exist, ensuring that the make process continues **WITHOUT** halting(停止) from the `rm` command or similar situations.

For example:

```
distclean: distclean-recursive
  -rm -f $(am_CONFIG_DISTCLEANFILES)
  -rm -f Makefile
```

`-rm -f $(am_CONFIG_DISTCLEANFILES)` attempts to remove the files specified in the variable `am_CONFIG_DISTCLEANFILES` forcibly, ignoring any errors due to the `-` prefix.
`-rm -f Makefile` attempts to remove the `Makefile` itself, also ignoring any errors.

This approach allows the distclean target to proceed through its recipe commands without stopping, even if some of the files it attempts to delete are not present, which is common in clean-up scenarios where files may or may not have been generated or already removed.

Macros

```
XYZ = foo1.o foo2.o foo3.o  # XYZ is a macro
foo: $(XYZ)
  g++ $(XYZ) -o foo
```

p.s. `g++` refers to GNU C++ compiler, which takes C++ source files (typically with .cpp, .cc, .cxx, .C or .c++ extensions) as input and produces an executable file as output.

Commonly you'll see people define `CC` macro for the compiler command (`gcc`) and `CFLAGS` macro for the compiler options(flags):

```
CC = gcc
CFLAGS = -O2 -g3 -Wall -Wextra

file.o: file.c
  $(CC) $(CFLAGS) -c somefile.c -o somefile.o
```

Phony Targets

`Phony` target is the target that does not represent files (like clean, which should be declared as `.PHONY: clean`) to tell `make` these targets aren't actual files. This prevents conflicts with existing files of the same name.

Since `Phony` target is the target that does not represent files, then it is also a rule that has **NO** output file.

For example:

```
.PHONY: clean

clean:
  rm -f *.o
```

`.PHONY clean` declares clean as a phony target.
`rm -f *.o` removes all object files in the current directory.
Since `clean` is phony, `make clean` will run this recipe regardless of whether a file named clean exists.

Running `Make`

Here's a template about how to run a `make` in the command line:

```
$ make target_name
```

`Make` is smart enough to see the dependencies (prerequisites) and avoid doing work that has already been done.

`Make` compares **timestamps**(the last modified times) of the target and its dependencies. If the target is older than any of its dependencies (meaning one or more dependencies have been updated since the target was last built), `make` will execute the commands associated with that target.

For example:

```
CC = gcc
foo: x.o y.o z.o
  $(CC) $^ -o $@
```

```
$ make foo
```

`Make` looks at the **timestamp** of `foo`(the target file) and the **timestamps** of its dependencies, `x.o`, `y.o`, `z.o`(the object files) to determine what files have changed since last run (last modified time of each file). If a file is up-to-date, it doesn't need to touch it (in this case, it does't need to link the three object files `*.o` together into `foo` again).

You can run a specific rule by specifying the target at the command line:

```
$ make z.o
```

You can override macros at the command line with similar `=` syntax:

```
$ make CC=clang
```

Common Mistakes

Often times, bugs arise from specifying the *dependencies* of the targets.

- A *missing* dependency means the product may be wrong. Make will falsely assume its job is done if it does not know whether needs to act on a certain file.
- An *extraneous* dependency is a more benign mistake. Files are still compiled the same way as before, but it may cause Make to do extra work.

Parallel Makes

Suppose you have a big project with multiple subdirectories:

```
project/
  Makefile
  subdir1/
    Makefile
  subdir2/
    Makefile
  subdir3/
    Makefile
```

One approach is to have Makefiles in each subdirectory. In this case, the **root** Makefile *delegate* works to the subsidiary Makefiles as:

```
test:
    for dir in subdir1 subdir2 subdir3; do
        (cd $$dir && make);
    done
```

- ➕ This modularizes the build process.
- ➖ However, the root Makefile is **sequential** code, and you don't have the opportunity to run multiple rules in parallel.

Another approach is to have one big Makefile at the project root will all the rules, including tests for anything under subdirectories. Nowadays, this is the more common pattern:

- ➕ It is a performance bonus, and it is usually automatically generated anyway.
- ➖ Extraneous/incorrect dependencies may slow down the process because there's less opportunity for parallelism. A missing dependency is even worse because they can cause different builds affecting different files, resulting in crashes.

Building Makefiles

In this section, we will illustrate how you can create a portable and adaptable **automated check** for C projects that might depend on some specific system features or libraries not universally available in your current development environment.

> For the following example, the C header `<stdckdint.h>` is not probably universally available yet, but you want to check its availability.
>
> The target of this example was to implement a simple, automated way to check for the presence of a hypothetical header file, `<stdckdint.h>`, and then compile a C program accordingly. This approach is especially useful in cross-platform software development, where the availability of certain features can vary widely between systems.

1. `configure` script

First, we'll write a `configure` shell script that checks if we can compile a C program that includes the `<stdckdint.h>` header.

```
#!/bin/sh

# Attempt to compile a test program
cat > checkfile.c << EOF
#include <stdckdint.h>
int main(void) { return 0; }
EOF

# Clean up previous configuration
rm -f config.h

# Check if the compilation succeeds
if gcc checkfile.c -o checkfile; then
  echo "#define HAVE_STDCKDINT_H 1" > config.h
else
  echo "#define HAVE_STDCKDINT_H 0" > config.h
fi

# Clean up test files
rm -f checkfile.c checkfile
```

This script will generate a `config.h` file, which our C program will include to know whether `<stdckdint.h>` is available.

> p.s. The `Heredoc` Syntax (inside the `<< EOF ... EOF`) is used to create a temporary C source file directly within the script. This file tests the availability of a specific feature, library, or header.
>
> In this case, it includes or utilizes something from <stdckdint.h> to check for its presence.
>
> If compilation succeeds, it's assumed the feature is available, and a define is added to `config.h`.

2. `C` program

Now, write a simple `C` program that uses features from `<stdckdint.h>` conditionally. Create a file named `main.c` with the following content:

```
#include "config.h"
#include <stdio.h>

int main(void) {
  #if HAVE_STDCKDINT_H
    printf("stdckdint.h is available.\n");
    // use the features for stdckdint.h
  #else
    printf("stdckdint.h is not available.\n");
  #endif
  return 0;
}
```

This allows your program to compile and run correctly in environments lacking specific features by providing alternative implementations or disabling certain functionality.

3. `Makefile`

Create a Makefile that compiles `main.c` into an executable file named `program`:

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

p.s. You can build the software follows these steps:

1. Run the `configure` script to generate `config.h`

   ```
   $ ./configure
   ```

2. Compile the `C` program using `make`:

   ```
   $ make
   ```

3. Run the compiled `C program`:

   ```
   $ ./program
   ```

This example demonstrates a simple, automated process for configuring a software build based on the availability of certain features or files. Real-world projects might use tools like `Autoconf` to generate configure scripts that handle many more conditions and configurations. This approach allows developers to write portable software that adapts to different environments and system capabilities.

Autoconf and Automake

Autoconf is a program that generates `./configure` files. Just as how Makefiles are automatically generated by `./configure`, the `./configure` file is generated by **Autoconf**. We see that build tools often build on top of each other, each generated by the next. This is a common pattern in software construction.

Automake is like a front-end abstraction for Autoconf, so like yet another level up from Autoconf.

Dependencies

You can think of them in terms of **dependency graphs**, which are typically (and ideally) *acyclic*.

Two ways dependencies come up in software construction:

Build-time Dependency

"File x depends on file y." This can be expressed simply in Makefile syntax:

```
x: y
    # create x via y, where y is the input
```

As a project scales, the dependency lists can get much longer.

```
foo.o: foo.c x.h y.h z.h
    gcc -c foo.c $^
```

Maybe you extend the list based on what you see from the `#include` lines in the source files.

This can be problematic because you need to make sure that the dependencies in the Makefile *match* those of the source code, else it would slow down your building at best and break your it if a dependency is missing.

Thus, instead of maintaining dependencies *by hand*, you should *calculate* them with some `make` preprocessor - a tool to automatically generate your Makefiles for you.

Packaging Dependency

Each package can be installed independently, BUT:

- Packages can depend on other packages
- Pnd every package can be version-specific.

This can get complicated, so ideally you'd want a **package manager**.

You can view the dependencies of a program, using `grep` as an example:

```
$ dnf deplist grep
Not root, Subscription Management repositories not updated
Last metadata expiration check: 1:16:41 ago on Mon 05 Dec 2022 04:57:02 PM
PST.
package: grep-3.1-6.el8.x86_64
  dependency: /bin/sh
   provider: bash-4.4.20-4.el8_6.x86_64
  dependency: /sbin/install-info
   provider: info-6.5-7.el8.x86_64
  dependency: libc.so.6()(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.14)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.2.5)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.3)(64bit)
```

```
    provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.3.4)(64bit)
    provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.4)(64bit)
    provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.5)(64bit)
    provider: glibc-2.28-211.el8.x86_64
  dependency: libpcre.so.1()(64bit)
    provider: pcre-8.42-6.el8.x86_64
  dependency: rtld(GNU_HASH)
    provider: glibc-2.28-211.el8.i686
    provider: glibc-2.28-211.el8.x86_64
```

And the dynamically linked libraries:

```
$ ldd $(which grep)
linux-vdso.so.1 (0x00007fff5ab88000)
        libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007f365a1fd000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f3659e38000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f3659c18000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f365a481000)
```

We notice that for example, `grep` requires `sh` because `egrep` utilizes a *shell script*. `grep` also needs the C library.