

Makefile

Compiling and Linking

Compiling and **linking** are two critical steps in the process of turning human-readable source code into an executable program. Here's an overview of what each step involves:

Compilation

Compilation is the process of transforming source code (written in languages like C, C++, etc.) into **machine code** or an intermediate format like **object code**. The compiler reads your source code and checks it for errors, syntax correctness, and other issues. If everything is correct, it translates the code into machine-level instructions that the computer's CPU can understand.

For a C program with the following source code file `main.c`:

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

The compiler (`gcc`, `clang`, etc.) will transform `main.c` into an object file (`main.o`), which contains compiled code but isn't yet executable.

- **Compilers** translate high-level code into lower-level object code.
- Object files are incomplete and need linking to be fully functional

Linking

Linking is the process that combines **object files** and **libraries** into a complete, runnable program. This step is necessary because most programs are made up of multiple parts, often spread across several files. Additionally, programs rely on external code from libraries (like the C standard library for functions like `printf`).

There are two types of linking:

- **Static linking:** The linker copies the required code from libraries into the executable itself.

- **Dynamic linking:** The program is linked to shared libraries (e.g., `.so` or `.dll` files) that are loaded at runtime. This reduces the executable size but requires the shared libraries to be available when the program runs.

The output of the linking step is an **executable file** (e.g., `a.out` on Unix-based systems or `my_program.exe` on Windows).

Suppose `main.o` needs to be linked with a standard library like `libc` to resolve the `printf` function. The **linker** combines `main.o` with the necessary libraries, producing an executable `my_program`.

- **Linkers** resolve references to external symbols (e.g., functions or variables) that are not defined within the same object file.
- The linker combines the necessary pieces to create a final executable program.

Here's a breakdown of how it works in sequence:

1. Compilation

- **Step 1:** Each source file is independently compiled into an object file.
 - The source code (e.g., `.c`, `.cpp` files) is translated into machine code by the compiler.
 - Each source file produces a corresponding object file (e.g., `.o` or `.obj`).
 - These object files are still incomplete because they often refer to functions or variables defined in other files or libraries.
 - **Output:** One or more object files (`.o`, `.obj`).

Example:

- `main.c` → `main.o`
- `foo.c` → `foo.o`

2. Linking

- **Step 2:** All the object files, along with any necessary libraries, are linked together to form an executable.
 - The linker combines the object files into a complete program, resolving references between them (like calling functions defined in different files).
 - It also pulls in any external libraries (such as the standard C library for functions like `printf`).
 - **Output:** A final executable (e.g., `my_program`, `a.out`, or `my_program.exe`).

Example:

- `main.o + foo.o + bar.o + standard libraries` → `my_program`

Why Is It a Two-Step Process?

This separation into two steps allows for **modular compilation**:

- You don't need to recompile all the files every time you make a small change to one file. Only the modified file needs to be recompiled, and then the linker combines the new object file with the existing ones to create the updated program.
- It supports the reuse of precompiled object files or libraries, making the build process faster and more efficient in larger projects.

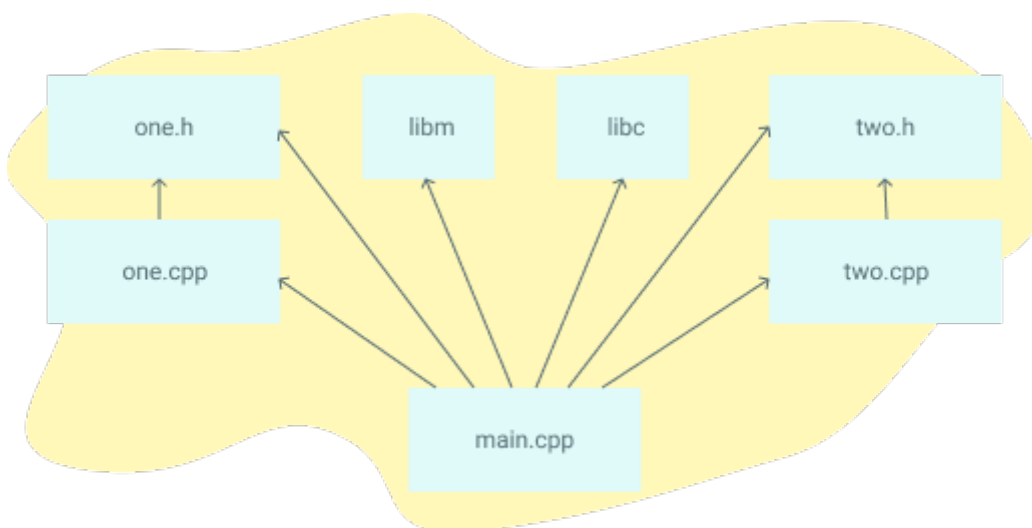
Why do Makefiles exist?

Makefiles automate this process by managing the compilation and linking of multiple files in large projects.

A **Makefile** is a special file used by the `make` utility to automate the build process of software projects. It defines a set of rules that dictate how to compile and link programs. When working on a project with multiple source files and dependencies, a Makefile can greatly simplify the build process by specifying:

1. **Targets:** These are typically files that you want to generate, like executables or object files. A target can also represent a task, like running tests or cleaning up compiled files.
2. **Dependencies:** Files or targets that must be up-to-date before a target can be built. For example, a `.c` file depends on a corresponding `.h` header file.
3. **Commands:** Shell commands to be executed to build the target. These typically include commands like `gcc` for compiling or `g++` for linking.

Here's an example dependency graph that you might build with Make. If any file's dependencies changes, then the file will get recompiled:



Interpreted languages like Python, Ruby, and raw Javascript don't require an analogue to Makefiles. The goal of Makefiles is to compile whatever files need to be compiled, based on what files have changed. But when files in interpreted languages change, nothing needs to get recompiled. When the program runs, the most recent version of the file is used.

However, it can automate tasks in other environments, like managing dependencies in a Python or JavaScript project.

Make file Basic Structure:

```
targets: dependencies
```

```
    ^command
```

```
    ^command
```

```
    ^command
```

Example Make file for Simple C++ project

```
# Compiler and flags
```

```
CC = gcc
```

```
CFLAGS = -Wall -g
```

```
# Target program
```

```
TARGET = my_program
```

```
# Source files
```

```
SRCS = main.c foo.c bar.c
```

```
# Object files (generated from source files)
```

```
OBJS = $(SRCS:.c=.o)
```

```
# Default target
```

```
all: $(TARGET)
```

```
# Rule to build the target program
```

```
$(TARGET): $(OBJS)
```

```
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
```

```
# Rule to build object files from source files
```

```
%.o: %.c
```

```
    $(CC) $(CFLAGS) -c $<
```

```
# Clean rule to remove generated files
```

```
clean:
```

```
    rm -f $(OBJS) $(TARGET)
```

Key Points:

- `$(CC)` and `$(CFLAGS)`: Variables for the compiler and flags.
- `$<` and `$@`: Special automatic variables where `$<` refers to the first dependency, and `$@` refers to the target.
- **clean target**: Commonly used to delete object files and executables after building.

Why Use Makefiles?

- **Efficiency**: Only recompiles files that have changed.
- **Automation**: Simplifies repetitive tasks like compilation, linking, or testing.
- **Portability**: Works across different systems without needing to write custom build scripts.

The Essence of Make

```
hello:
    echo "Hello, World"
    echo "This line will print if the file hello does not exist."
```

- We have one *target* called `hello`
- This target has two *commands*
- This target has no *prerequisites*

We'll then run `make hello`. As long as the `hello` file does not exist, the commands will run. If `hello` does exist, no commands will run.

It's important to realize that I'm talking about `hello` as both a *target* and a *file*. That's because the two are directly tied together. Typically, when a target is run (aka when the commands of a target are run), the commands will create a file with the same name as the target. In this case, the `hello` *target* does not create the `hello` *file*.

Let's create a more typical Makefile - one that compiles a single C file. But before we do, make a file called `blah.c` that has the following contents:

```
// blah.c
int main() { return 0; }
```

Then create the Makefile (called `Makefile`, as always):

```
blah:
    cc blah.c -o blah
```

This time, try simply running `make`. Since there's no target supplied as an argument to the `make` command, the first target is run. In this case, there's only one target (`blah`). The first time you run this, `blah` will be created. The second time, you'll see `make: 'blah' is up to date`. That's because the `blah` file already exists. But there's a problem: if we modify `blah.c` and then run `make`, nothing gets recompiled.

We solve this by adding a prerequisite:

```
blah: blah.c
cc blah.c -o blah
```

When we run `make` again, the following set of steps happens:

- The first target is selected, because the first target is the default target
- This has a prerequisite of `blah.c`
- Make decides if it should run the `blah` target. It will only run if `blah` doesn't exist, or `blah.c` is *newer than* `blah`

This last step is critical, and is the **essence of make**. What it's attempting to do is

- Decide if the prerequisites of `blah` have changed since `blah` was last compiled. That is, if `blah.c` is modified, running `make` should recompile the file.
- Conversely, if `blah.c` has not changed, then it should not be recompiled.

It uses the filesystem timestamps as a proxy to determine if something has changed. File timestamps typically will only change if the files are modified.

SOURCE:

<https://makefiletutorial.com/>

Revision #8

Created 7 October 2024 23:40:01 by victor

Updated 12 October 2024 00:12:47 by victor