Lecture 02 - Separate Compilation and Make (last updated: 2023-09-11)

C is a compiled language, meaning we cannot run it directly; we must first compile C code to an executable file before running the executable file.

Compilers translate code that you write into machine code that computer hardware can run. They provide several benefits for programmers:

- Portability: the same source code can be compiled for different hardware
- Readability: source code is easier for humans to read, maintain, and write
- Safety: compilers can prevent errors, e.g., by catching scope or type errors

Here is an example of compiling and running a hello world C program:

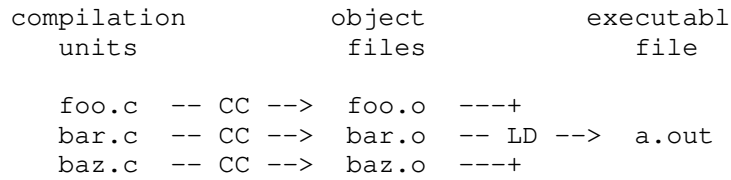
```
$ cat hello.c           # show contents of hello world program
#include <stdio.h>

int main(int argc, char **argv) {
    printf("%s\n", "Hello, world!");
    return 0;
}
$ gcc hello.c           # compile and link hello world...
$ ls                    # ...creating an executable file named a.out
a.out   hello.c
$ ./a.out               # execute a.out
Hello, world!
```

Note that for historical reasons, the default executable name is a.out. We will later see how to customize the executable name.

Separate compilation

Compilers are complex pieces of software that require lots of computing power. Old computers took a long time to compile large software projects, so compilation was broken down into "compilation units" that could be compiled incrementally and independently. Each compilation unit is compiled into an "object file"; object files are linked together to produce an executable file.



C and many other compiled languages still use separate compilation today. Separate compilation also allows programmers to link object files compiled from different languages.

Some naming conventions:

- cc: the command used to invoke the C compiler
- ld: the command used to invoke the linker
- .c: the file extension typically used for compilation units written in C
- .o: the file extension typically used for object files

Nowadays we use gcc (GNU Compiler Collection) as both a C compiler and a linker.

We seldom use the cc and ld commands on their own; on modern Linux machines, those commands just invoke GCC under the hood.

Here we use GCC to only compile (but not link) hello.c, using the -c flag:

```
$ gcc -c hello.c    # compile hello.c...
$ ls                # ...producing hello.o
hello.c   hello.o
```

And now we link the hello.o object file:

```
$ gcc hello.o       # link hello.o...
$ ls                # ...producing a.out
a.out   hello.c   hello.o
```

Note that we must link hello.o even if it is the only object file in the executable; object files are not executable on their own.

Useful compiler flags

When we invoke GCC, we may specify additional flags to customize its behavior. For instance, we may compile with the following flags:

```
$ gcc -g -Wall -Wpedantic -std=c17 -c -o hello.o hello.c
```

Here is what they do:

- -g: include debugging info (which we will use later in the course)
- -Wall: enable all warnings (useful for catching bugs)
- -Wpedantic: enforces the C language standard
- -std=c17: specifies the C17 language standard
- -c: compilation only
- -o hello.o: use hello.o as the output filename

These flags may be passed to gcc in any order. -o hello.o is not actually necessary since it just specifies the default object filename, but it is good practice to specify the output filename explicitly.

We can also use the -o flag when linking, to set the output filename to hello instead of a.out:

```
$ gcc -o hello hello.o  # link hello.o...
$ ls                    # ...producing hello
hello   hello.c   hello.o
```

Linkers and symbols

The linker (ld) links together object files, which define symbols. Symbols are the names of functions and global variables shared between compilation units. When an object file uses a symbol defined in another compilation unit, it leaves a placeholder for that symbol.

The linker's job is to fill each symbol placeholder with a reference to that symbol's single definition.

Consider mymod.c, which defines the symbol "mod":

```
// mymod.c

// Define the "mod" symbol
int mod(int x, int y) {
    return x % y;
}
```

Now consider myprogram.c, which defines the symbol "main":

```
// myprogram.c

// Define the "main" symbol
int main(void) {
    return mod(4, 2);
}
```

When we compile myprogram.c to produce myprogram.o as follows,

```
$ gcc -g -Wall -Wpedantic -std=c17 -c -o myprogram.o myprogram.c
```

myprogram.o will contain a placeholder for the symbol "mod". The symbol "mod" was used in myprogram.o -- the main() function called mod() function -- and the symbol "mod" was defined in another object file, mymod.o. The linker will resolve the placeholder with the definition at link time:

```
$ gcc myprogram.o mymod.o -o myprogram
```

By the way, if you had compiled myprogram.c shown above, you would have gotten a warning from the compiler like this:

```
warning: implicit declaration of function ’mod’
```

The C language requires that a function be either fully defined or at least declared before it is called. A function can be declared by writing its prototype as follows:

```
// myprogram.c

// Declare (but not define) the mod() function
int mod(int x, int y);

// Define the "main" symbol
int main(void) {
    return mod(4, 2);
}
```

A function declaration (aka prototype or signature) is therefore the return type, the function name, and the parameter types, followed by a semicolon.

The "extern" keyword is implicit for function declarations; the following declarations are equivalent:

```
int mod(int x, int y);

extern int mod(int x, int y);
```

In summary, the linker has three responsibilities:

- ensure that all symbols are defined at most once

- ensure that there are no missing definitions
- ensure that the "main" symbol is defined

Headers and #include

If multiple C files contain a call to a function add(), for example, manually adding the add() function's prototype at the top of all the files that use it would be not only cumbersome, but also error-prone.

Header files help us solve this problem. Conventionally, they end with a ".h" extension, and contain any number of declarations that we wish to share between compilation units. For example, consider the header file myadd.h:

```
// myadd.h

int add(int x, int y);
```

This header file ensures that the same declaration of "add" is seen by every compilation unit. We use a C preprocessor directive, #include, to copy the contents of myadd.h into compilation units that use "add", e.g., this version of myprogram.c:

```
// myprogram.c

#include "myadd.h"      // Copy in the textual contents of myadd.h

int mod(int x, int y);

int main(void) {
    return mod(add(2, 2), 2);
}
```

Note that #include (and other C preprocessor directives, which begin with "#") perform purely textual modifications to .c files. We could have replicated the behavior of #include "myadd.h" by manually pasting in the contents of myadd.h.

Standard C Library

Consider the following version of myprogram.c:

```
// myprogram.c

#include "myadd.h"  // myadd.h is searched in the current directory
#include <stdio.h>  // stdio.h is searched in the system directory

int main(void) {
    printf("%d\n", add(2, 3));
    return 0;
}
```

The content of stdio.h, which gets copied into myprogram.c when it gets compiled, will contain the prototype of the printf() function. The stdio.h file collects the prototypes of input & output related functions (among other things) defined in a set of pre-installed object files called the Standard C Library.

It is important to note that stdio.h file does NOT contain the definition of the

printf() function. It only contains the declaration. The definition of the function is found in an object file belonging to the Standard C Library, which gets automatically linked in by the linker, without you having to specify the object file name in the linking command.

Function types

Object files only record the name, but not the type, of each symbol they contain. For functions, this means that only their names are recorded, not their parameter and return types.

Consider linking myprogram.c above (in the Standard C Library section) with the following new version of myadd.c with a modified definition of add() that takes three parameters:

```
// myadd.c

int add(int x, int y, int z) {
    return x + y + z;
}
```

Each file compiles independently without warning. And you can link myprogram.o and myadd.o without any warning as well. But when we run the executable, it will produce unpredictable results. (Try it!)

We can fix it by adding #include "myadd.h" into myadd.c as well:

```
// myadd.c

#include "myadd.h"

int add(int x, int y, int z) {
    return x + y + z;
}
```

Now the compiler will complain when it sees that the type of the declaration does not agree with the definition.

Make

Because myprogram.o is produced by compiling myprogram.c, we say that myprogram.o depends on myprogram.c. Similarly, because myprogram is produced by linking myprogram.o, myadd.o, and mymod.o, we say that it depends on those object files.

We should only need to rebuild something if one of the files it depends on has changed; after all, part of the reason for separate compilation is to avoid the cost of rebuilding things we do not need to rebuild.

Make is a build system that helps us manage incremental builds. You can find its manual online here: https://www.gnu.org/software/make/manual/make.html

Make runs commands for you according to instructions specified in a Makefile. For example, it can help you run any of the GCC build commands shown above. A Makefile should specify one or more "rules" for building files; each rule looks like this:

```
    myprogram: foo.o bar.o baz.o
            gcc -o myprogram foo.o bar.o baz.o
```

where:

- myprogram is the name of the "target," i.e., the file this rule will build
- foo.o, bar.o, and baz.o are the "prerequisites" that the target depends on
- gcc <etc.> is the "recipe," i.e., the shell command that builds the target

Make uses the "last modified" timestamp of each file to figure out if a target needs to be rebuilt. Make will rebuild a target if it doesn't exist, or if it was last modified earlier than one of its prerequisites.

Putting it all together

myadd.h:

```
    #ifndef _MYADD_H_
    #define _MYADD_H_

    int add(int x, int y);

    #endif
```

myadd.c:

```
    #include "myadd.h"

    int add(int x, int y) {
        return x + y;
    }
```

mymod.c:

```
    int mod(int x, int y) {
        return x % y;
    }
```

myprogram.c:

```
    #include <stdio.h>
    #include "myadd.h"
    int mod(int x, int y);

    int main() {
        printf("add(3, 4) returned: %d\n", add(3, 4));
        printf("mod(17, 5) returned: %d\n", mod(17, 5));
        return 0;
    }
```

Makefile

```
    myprogram: myprogram.o myadd.o mymod.o
```

```
        gcc myprogram.o myadd.o mymod.o -o myprogram

myprogram.o: myprogram.c myadd.h
        gcc -c -o myprogram.o myprogram.c

myadd.o: myadd.c myadd.h
        gcc -c -o myadd.o myadd.c

mymod.o: mymod.c
        gcc -c -o mymod.o mymod.c
```

myprogram.c and myadd.c both #include the myadd.h header file, so the object files they compile to both depend on the myadd.h. That is, if, we modify myadd.h, we should update myprogram.o and myadd.o to make sure they are compiled with our latest modifications to myadd.h.

Here is the output of make:

```
$ make
gcc -c -o myprogram.o myprogram.c
gcc -c -o myadd.o myadd.c
gcc -c -o mymod.o mymod.c
gcc myprogram.o myadd.o mymod.o -o myprogram
```

Make prints out the commands that it runs. Run ls to confirm that these commands successfully produced myprogram.o, myadd.o, mymod.o, and myprogram.

Note that Make builds myprogram because that is the first target in the Makefile; it builds myprogram.o, myadd.o, and mymod.o because they are dependencies of myprogram.

If you run make again, it won't do anything, since the timestamp of myprogram is later than its dependencies (the .o files).

```
$ make
make: 'myprogram' is up to date.
```

We can use the touch command (man touch) to update the timestamp of a dependency and trick Make into rebuilding myprogram:

```
$ touch myadd.c     # update time stamp of myadd.c
$ make
gcc -c -o myadd.o myadd.c
gcc myprogram.o myadd.o mymod.o -o myprogram
```

Sample Makefile

Make includes all sorts of useful features like variables, implicit rules, and phony targets. Here is the same Makefile for myprogram, but with a ton of comments explaining the essential features of Make.

```
# This Makefile should be used as a template for future Makefiles.
# It's heavily commented, so hopefully you can understand what each line does.
```

```
# Makefiles allow us to define variables that can be expanded elsewhere.
#
# Some variables, like CC, CFLAGS, LDFLAGS, and LDLIBS, are used to customize
# the behavior of Make's built-in rules.

# Set CC to gcc to use gcc as our C compiler
CC = gcc

# Compilation options:
# -g: include debugging info symbols
# -Wall: enable all warnings
# -Wpedantic -std=c17: enforces the C17 language standard
CFLAGS = -g -Wall -Wpedantic -std=c17

# Linking options:
LDFLAGS =

# List the libraries you need to link with in LDLIBS.
# For example, use -lm for the math library.
LDLIBS =

# The first target gets built when you run make.
# It's usually your executable (myprogram in this case).
#
# Note that we did not specify the linking recipe.
# Instead, we rely on one of Make's implicit rules:
#
#     $(CC) $(LDFLAGS) <dependent-.o-files> $(LDLIBS) -o <executable-name>
#
myprogram: myprogram.o myadd.o mymod.o

# myprogram.o depends not only on myprogram.c, but also on myadd.h, which it
# includes.  myprogram.o will be recompiled if it has an earlier timestamp than
# any of its dependencies.
#
# Make uses the following implicit rule to compile a .c file into a .o file:
#
#     $(CC) -c $(CFLAGS) -o <target-.o-file> <the-.c-file>
#
# Make's implicit rule assumes myprogram.o depends on myprogram.c, so we can
# omit myprogram.c in the dependency list if we want to.
myprogram.o: myprogram.c myadd.h

# myadd.o depends on myadd.c and myadd.h.
myadd.o: myadd.c myadd.h

# mymod.o depends on mymod.c.
mymod.o: mymod.c

# Always provide the "clean" target that removes build artifacts (e.g., the
# target executable and .o files) and other garbage that may be created during
# the development process.
#
# The "clean" target does not correspond to a filename, so we tell Make that
# it's a "phony" target, meaning it does not need to check the timestamp of
# a file called "clean".
.PHONY: clean
clean:
	rm -f *.o a.out core myprogram
```

```
# The "all" target can be useful if your Makefile builds multiple programs.
# Here we'll have it first do "clean", and rebuild the myprogram target.
#
# Like "clean", we declare it as a phony target because it doesn't actually
# build a file named "all".
#
# Dependencies are built in the order they are declared, left to right.
.PHONY: all
all: clean myprogram
```