Lecture 03 - Separate Compilation and Make (last updated: 11/02/22)

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:

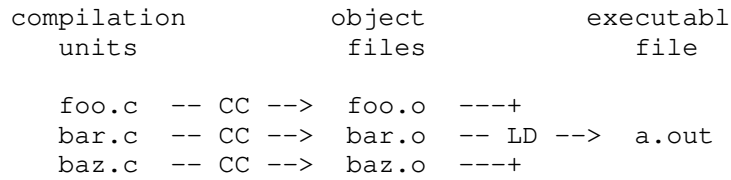
```
$ cat hello.c           # show contents of hello world program
#include <stdio.h>

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
```

Make

Because hello.o is produced by compiling hello.c, we say that hello.o depends on hello.c. Similarly, because a.out (from the separate compilation diagram) is produced by linking foo.o, bar.o, and baz.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.

Make includes all sorts of useful features like variables, implicit rules, and phony targets. Please see the sample Makefile accompanying these lecture notes for details and examples.

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", and declares a placeholder for the symbol "mod":

```
// myprogram.c

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

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

In C, top-level (global), non-static definitions turn into symbol definitions;

top-level declarations turn into placeholders.

The same can be done for any type of variable, not just functions:

```
// Declare (but not define) the "x" symbol
extern int x;

// Define the "x" symbol with the value 42
int x = 42;

// Alternatively, define the "x" symbol with the default/uninitialized value
int x;
```

Note that 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

Manually declaring all the symbols you use is not only cumbersome, it's also error-prone! Object files only record the name but not the type of each symbol you declare, so it's up to you to make sure that the type of each symbol declaration matches up with the type of its definition.

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.

We also #include "myadd.h" in the compilation unit that defines "add":

```
// myadd.c

#include "myadd.h"

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

Now the compiler will complain if it sees that the type of the declaration does not agree with the definition---something the linker cannot do, because the linker does not know about types!

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. We can write an appropriate Makefile for this build plan:

```
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
```

Note that even though myadd.h is a prerequisite of myprogram.o and myadd.o, we do not write myadd.h in the recipe to compile each of those object files. We declare that dependency to ensure that we recompile myprogram.o and myadd.o if there is any modification to myadd.h.