Multiple source files Makefile

Week 7

Multiple source files

A program can be split up into multiple files. This makes it easier to edit and understand, especially in the case of large programs—it also allows the individual parts to be compiled independently.

A C-program always contains the `main` function, which acts as an interface to the operating system. The command line arguments are handled as `main` function arguments.

Functions need to be declared before they are called, otherwise the code will not compile successfully.

We need to use ***header files*** to do this declarations.

Files. `Example1 > main.c, hello_func.c, hello.h`

main.c

```
#include "hello.h"
int main (void)
{
  hello("world");
  return 0;
}
```

`main.c`

We call a new ***external function*** `hello`, which is defined in a separate file 'hello_func.c'.

The main program also includes the header file `hello.h` which contains the *declaration* of the function `hello`.

Declaration tells the compiler about a function's name, return type, and parameters.

We do not need to include the system header file `stdio.h` in `main.c` to declare the function `printf`, because `main.c` does not call `printf` directly.

hello.h

The declaration in 'hello.h' is a single line specifying the prototype of the function hello:

```
void hello(char name[]);
```

hello_func.c

```
#include <stdio.h>


void hello (char name[])
{
  printf ("Hello, %s!\n", name);
}
```

Compiling

Note the difference between the two forms of the include statement
`#include "FILE.h"` and `#include <FILE.h>`

`#include "FILE.h"` searches for `FILE.h` in the current directory before looking in the system header file directories.

`#include <FILE.h>` searches the **system** header files, but does not look in the current directory by default.

To compile these source files with gcc:

```
gcc hello_func.c main.c -o prog
```

Object files

Using the command `gcc hello_func.c main.c -o prog` will recompile every source file every time, even if a change is just being made to one source file.

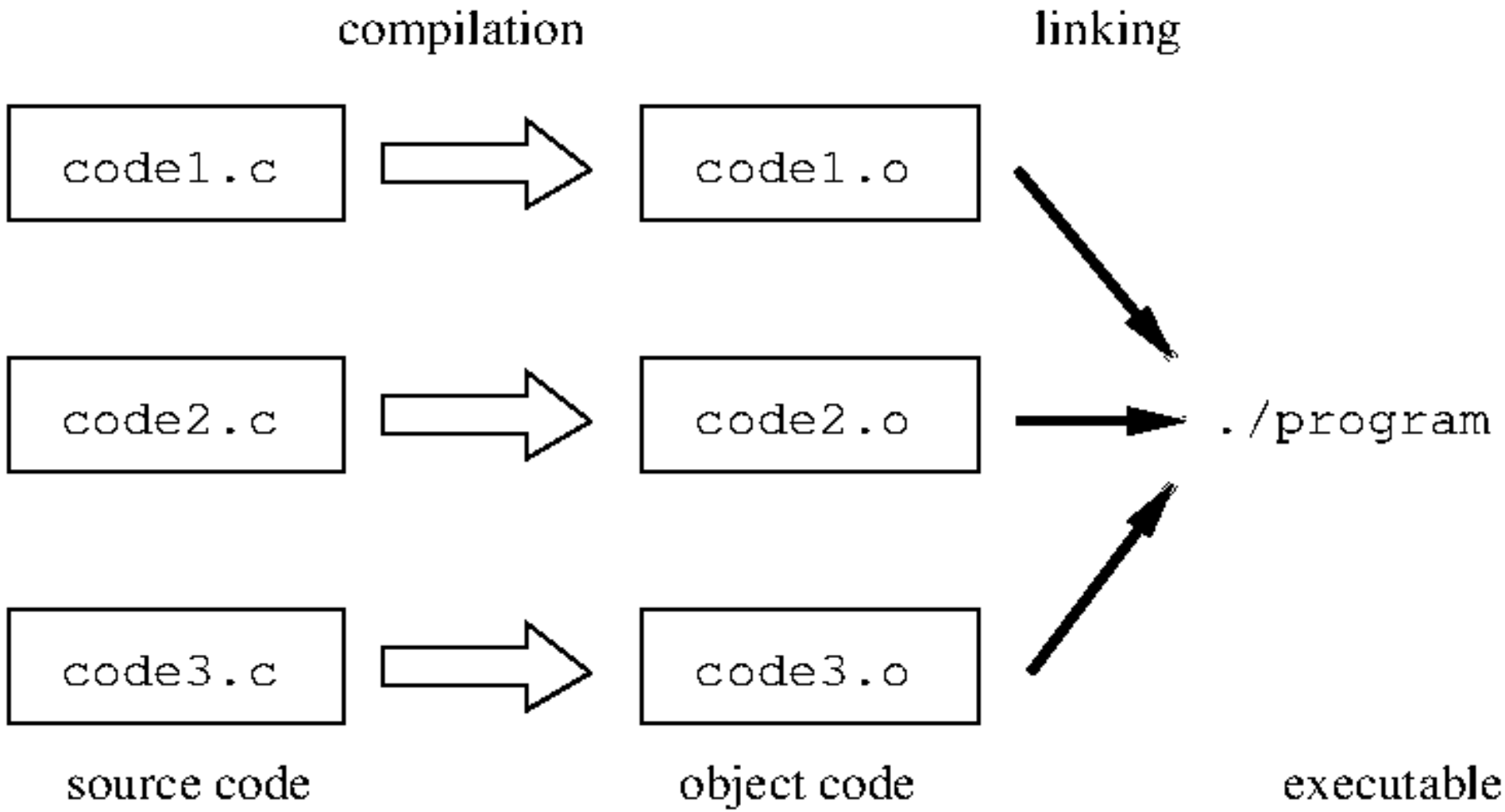
If a program uses very many source files and some of them are very large, then this can be very slow; this is because every time the compiler is being run, ***it is recompiling every source file***, even the source files which have not been changed since the last compile.

One way to resolve this issue is to use object files, which contain ***object code***. Under the hood, every time the compiler is being run, it first creates object code for each source file, and then it links all the object code together into a single executable file.

Different files

- **Source code** refers to high level code which is generated by a programmer.
- **Object code** refers to low level code which is understandable by machine. Object code is generated from source code after going through compiler. The `-c` flag can be used to tell the compiler to just create an object file, but not to link anything together into an executable.
- **Executable** (also called the binary) is the output of a linker after it processes the object code.

Multiple sources



Compiling → linking → executing

Linking refers to the creation of a single executable file from multiple object files:

First create the object files:

```
gcc hello_func.c –c
gcc main.c -c
```

Then link these to an executable:

```
gcc main.o hello_func.o -o program
```

Now we can run the executable:

```
./program
```

Example

Files: Example 2 > `main.c,prod.c,prod.h,sum.c,sum.h`

```
gcc main.c -c
gcc sum.c -c
gcc prod.c -c

gcc main.o sum.o prod.o -o program
```

Guards

Note that header files may be included in the source file multiple times, leading to compiler errors.

If the provided identifier (`PROD_H`) does not exist, this includes the code between it and the closing `endif`.

```
#ifndef PROD_H    /* Include guard */
#define PROD_H
-- -
#endif
```

Makefile

- Makefiles provide a way of automating the build process of programs with multiple source files; instead of manually telling the compiler to recompile the object code for each source file that has been updated since the last build, and then telling the compiler to link everything together into the executable file, a makefile can be used to automatically decide which object files to recompile by comparing the time-stamp of those object files with the source files which generate them.

- A makefile consists of one or more rules, and each rule consists of a target, one or more dependencies, and a command. Each rule starts off with the target, then a colon and a space, then the list of dependencies, seperated by spaces, then a new line, then a tab character (it **must be a tab character** and not just several spaces, otherwise the makefile won't work), then the command.

Makefile

Below is an example of a rule:

```
target: component1 component2 component3 . . .
	command1
	command2
	command3
```

Make can be used to manage any project where some files must be updated automatically from others whenever the others change (dependencies)

Makefile (Example1)

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

To run this makefile, save it in a file called `makefile` in the same directory as the .c source files, and then run the command make in terminal.

Makefile (Example2)

```
program: main_2.o sum.o prod.o
    gcc main_2.o sum.o prod.o -o program
main_2.o: main_2.c
    gcc main_2.c -c
sum.o: sum.c
    gcc sum.c -c
prod.o: prod.c
    gcc prod.c -c
```

`Strtok` (for project)

- Breaks a string into a series of tokens using the given delimiter.
- For example, the data in project work looks like this:

02.01.2020 22:00;1;10615801;0;1978337;1269010;2799999;2660199;202531

We can split that into tokens by using `strtok` and delimiter “;”

File : `Example3.c`

On the first call, the function expects a string as argument. In subsequent calls, the function expects a ***null pointer*** and uses the position right after the end of the last token as the new starting location for scanning.

Compiler options

- `gcc -Wall` enables all compiler's warning messages. This option should always be used, in order to generate better code
- `gcc -pedantic` enables the compiler to generate warnings if your code uses any language feature that conflicts with strict ISO C or ISO C++.