Project organisation and extern libraries

Variables

Lecturers: Mirko Jantschke, Pascal Scholz

4. Dezember 2018

Contents

Header files

It is a mess

The current version of our Dungeon is *2_finished/main.c*. In this file, there are:

- 205 lines of code
- 4 includes
- 2 custom enums
- 3 custom data type definitions
- 4 global variables
- 14 functions and its prototypes
- And the main function

We now will organize the project in multiple files.

Header files

The first thing we do is to seperate the I/O part into a file called *io.c*:

- *handle_input, draw_board, print_entity*

Since we are calling the I/O functions in our *main.c* file, the prototypes have to stay accessible from there.
This is why we are creating our own **header file** *io.h*, containing the prototypes, and we include it in our *main.c* and *io.c*:

```
1 #include "io.h"
```

- Note that we use **"..."** for own includes instead of $< \ldots >$.
- The quotation marks contain the relative path to the header file.

Data structures

The next step is to seperate all functions concerning the custom data structures to *datastructures.c*:

- init_entity
- init_monster_list
- free_monster_list
- add_monster
- monster_list_map
- position_covered
- has_position

We put the type definition of our structs with the prototypes into the header file *datastructures.h*.

Global variables

Since the source files all are accessing the global variables, we move them into another header file *globals.h*.

Finally we outsource the function *out_of_bounds* and the global variable *next_coords* into *board.c*.

- Try to find the remaining includes by yourself.

To avoid multiple includes of one header file we can, and should, do as follows:

```
1 #ifndef BOARD_H
2 #define BOARD_H
3
4 /* header file content here */
5
6 #endif /* BOARD_H */
```

Where to put these files?

Now we have a mess of **.h** and **.c** files hanging around in a single directory.
Best practice is to:

- Put all header files into an *inc* or *include* directory
- Put all source files into a *src* or *source* directory

Where to put these files?

Now we have a mess of **.h** and **.c** files hanging around in a single directory.
Best practice is to:

- Put all header files into an *inc* or *include* directory
- Put all source files into a *src* or *source* directory

But wait! Now we have to write awful relative paths into every file:

```
1 #include "../include/foo.h"
2
```

Where to put these files?

Now we have a mess of **.h** and **.c** files hanging around in a single directory.
Best practice is to:

- Put all header files into an *inc* or *include* directory
- Put all source files into a *src* or *source* directory

But wait! Now we have to write awful relative paths into every file:

```
1 #include "../include/foo.h"
2
```

You can pass additional include paths to gcc with the **-I** flag:

```
1 #include <foo.h>
2
```

```
1 $ gcc src/foo.c -I./include
2
```

Make

Compiling

To compile a project from multiple files you have to pass all source files to gcc:

```
$ gcc -Wall -o main datastructures.c io.c board.c main.c
```

- Note that you do not have to add the header files.
- They do not need to be compiled separately.

Makefiles

There is a package called *make* that allows you to write the compile order into a file and start compiling with a more simple command:

```
$ make
```

To use it, you have to install the package and create a file named *Makefile* in your project directory.

- Note that the **make** package is much more mighty than that.

Make rulez!

You can define rules in your makefile:

```
all:
    gcc -o main datastructures.c io.c board.c main.c
debug:
    gcc -Wall -g -o main datastructures.c io.c board.c main.c
```

Now you can build your project with:

```
$ make all
```

or with

```
$ make debug
```

if you want to have errors and debug information.

- Note: if you are calling make without a target, the first one is built.

About compiling

With many source files, we do not want to recompile the whole project if we changed a tiny piece of code.
Make offers the opportunity to compile only if dependencies have changed. But first a little theory about compiling:

process	input	output	notes
preprocessor	**.c*	source code with substitutions **.i*	
compiling	**.i*	assembler mnemonics **.s*	debug information is added
assembling	**.s*	byte code **.o*	not executable yet
linking	**.o*	executable byte code * / **.exe*	

We would like to split the compilation process and keep the unlinked *.o files to link newly build *.o files with existing ones.

Dependencies

gcc compiles without linking when the -c flag is set.
You can add a dependency to your makefile by writing it behind the rule identifier. Dependencies on other rules are allowed:

```
all: firstfile.o secondfile.o
    gcc -o main firstfile.o secondfile.o

firstfile.o: firstfile.c secondfile.h
    gcc -c firstfile.c

secondfile.o: secondfile.c secondfile.h
    gcc -c secondfile.c
```

- Note that both source files have the same header file as a dependency here.

A few tutorials later

```
1  CC       =gcc
2  SRCDIR   =src
3  OBJDIR   =build
4  INCDIR   =inc
5  EXE      =spaceinvaders
6  SRC      =$(wildcard src/*.c)
7  DEP      =$(SRC:%.c=%.d)
8  OBJ      =$(SRC:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
9  INC      =-I./$(INCDIR)/
10 CFLAGS   =-std=c99 ${INC} -Wall -Wextra -Wpedantic -
       Werror
11 LDFLAGS =-lncurses
12 VPATH    =$(SRCDIR)
13
14 .PHONY: all clean debug
15
16 all: $(OBJ) $(EXE)
17
18 debug: CFLAGS += -g
19 debug: $(EXE)
20
```

```
20 $(OBJ): | $(OBJDIR)
21 $(EXE): $(OBJ)
22     $(CC) -o $@ $^ $(LDFLAGS)
23
24 -include $(DEP)
25
26 %.d: %.c
27     $(CC) -MM ${INC} $*.c > $*.d
28     sed -i -e 's|.*:|$(patsubst $(SRCDIR)/%,$(
          OBJDIR)/%,$*).o:|' $*.d
29
30 build/%.o: %.c
31     $(CC) -c $(CFLAGS) -o $@ $<
32
33 clean:
34     rm -f $(EXE) $(OBJ) $(DEP)
35     rm -rf $(OBJDIR)
36
37 $(OBJDIR):
38     mkdir $@
39
```

Other build tools

make is a very low-level tool, not restricted to C/C++,

- which is great, because it can be used for anything where certain commands have to be used to generate files,
- which is bad, because make does not know anything about C/C++ and offers only a rudimentary framework to execute commands.

Other options

- Autotools
- CMake
- Scons
- ...

CMake

CMake stands for cross-platform make and offers a more high-level, declarative approach to building.

CMake does not build projects itself, instead it generates the necessary project files for other tools such as Make, Visual Studio or XCode.

```
cmake_minimum_required (VERSION 2.8.0)
project (MyProject)
set(CMAKE_BUILD_TYPE "Release")
list(APPEND CMAKE_C_FLAGS "-std=c11 -Wall -Werror")
target_include_directories (${CMAKE_CURRENT_SOURCE_DIR})
add_library (mylib lib.c)
add_executable (myprogram main.c foo.c)
target_link_libraries (myprogram mylib)

```

ncurses

An external library

The next step to build an awesome ASCII Dungeon is to get rid of the ënter mashingäfter entering the direction.

There is no way to do this cross platform with the standard library. That is why we will use an external one: **ncurses**.

We prepared this for you in *3_finished/io.c*. If you want, you can try to understand the content of that file.

- Have a closer look at `https://de.wikibooks.org/wiki/Ncurses`