Modules

Design

Big problems need to be broken down into smaller subproblems

Big *programs* need to be broken down into smaller pieces - modules

It pays to take some time to design the modules, e.g. by scribbling on paper, before programming

Functions

Suppose we have a two-function program, defined and compiled like this:

program.c

```c
void print() { ... }
int main() { ... }
```

```
gcc -std=c99 program.c
```

Multiple modules

Instead, we can put the functions in two files, and compile like this:

print.c
```
void print() { ... }
```

main.c
```
void print();
int main() { ... }
```

```
gcc -std=c99 main.c print.c
```

Each module needs declarations of the functions it uses from other modules

Signatures and Types

Suppose we create a module with several functions

Other modules need to include declarations of those functions

The same goes for declarations of types created by the module

We don't want to repeat the list of declarations in every module

Header files

So, make a header file for each module with the module's declarations in it

print.h
```c
void print();
```

print.c
```c
void print() { ... }
```

main2.c
```c
#include "print.h"
int main() { ... }
```

```
gcc -std=c99 main2.c print.c
```

Own header

A good thing to do is for a module to include its own header file:

```
#include "print.h"
#include <stdio.h>

void print() { ... }
```

Then the compiler checks that the declarations in the header match the definitions in the `.c` file

Organisation

Two different designs will often be implemented in the end using quite similar functions

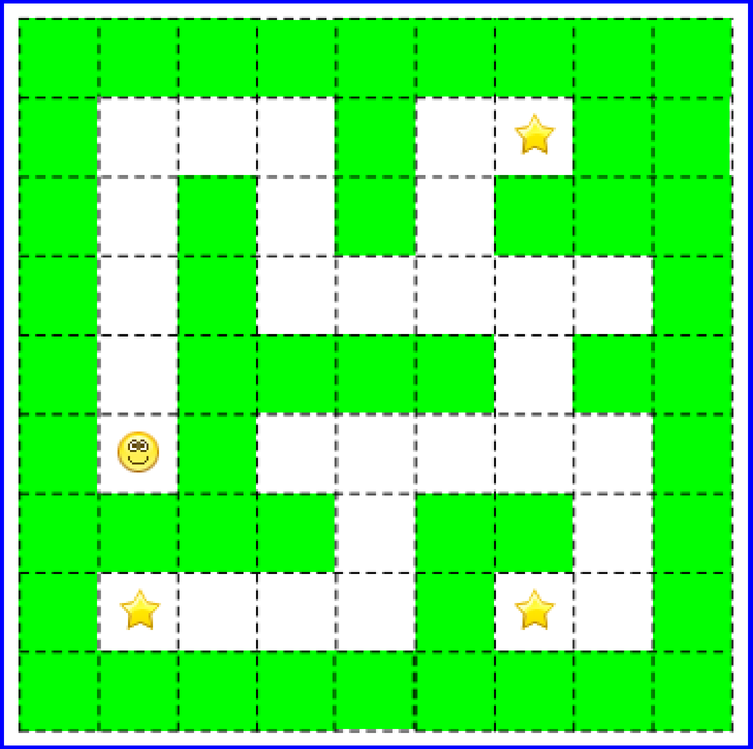
What's different is the organisation of the functions

It turns out, for all programs except the tiniest, that the organisation of the functions is just as important as the functions themselves

The issues are the ease of development, and the ease of automatic testing

Example: a grid game

Imagine a graphical grid-based game, e.g a maze



The grid containing blank spaces, walls, a player, and some stars to collect

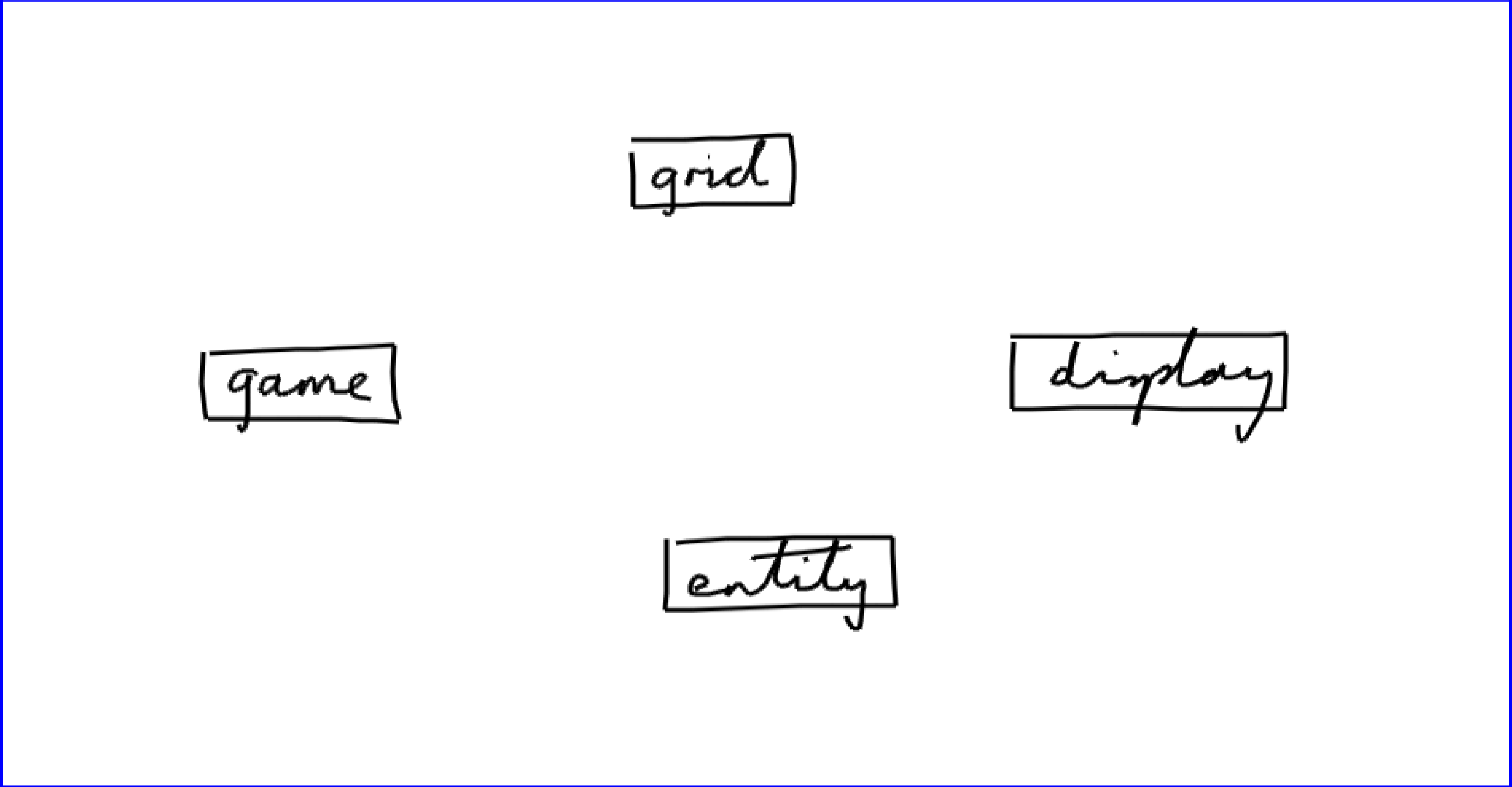
Sketch

Let's do a rough sketch of some possible modules

A reasonably sensible split might be an overall game control module, a grid module to keep track of where everything is, an entity module for the behaviours of the individual things in the grid, and a display module to show the game on screen

Module sketch

So we'll need, maybe, modules like this



Dependencies

Next, we want to work out which modules depend on which others

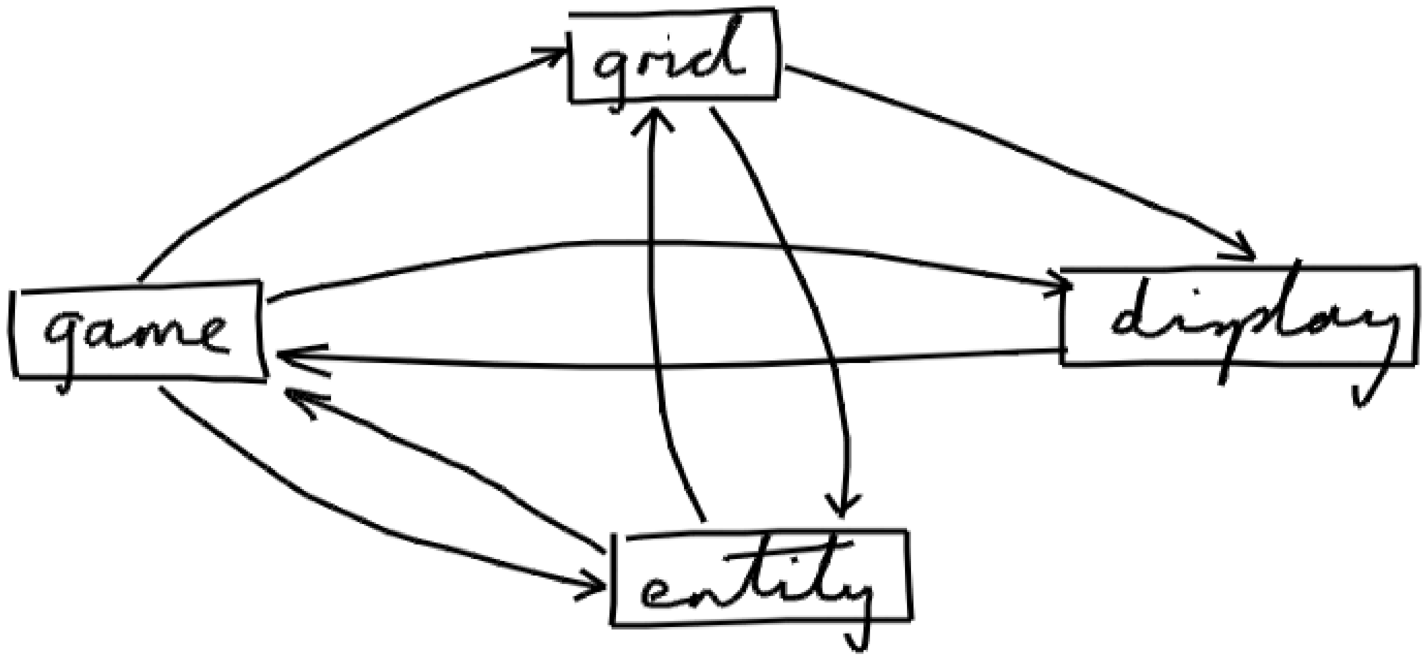
A module A *depends on* a module B if a function in module A calls a function in module B

We can picture it by drawing an arrow from A to B

Working out dependencies in advance needs experience, because you have to imagine the function calls needed in the implementation

Dependency sketch

It is easy to imagine dependencies like this:



A tangled mess

The sketch that we've drawn has a general problem and a specific problem

The general problem is that it is a tangled mess

If modules have a mess of dependencies between them, there is ***no gain***, compared to just putting all the functions in one file

And that limits the size of a project before it gets out of hand and becomes unmaintainable

Cyclic dependencies

The more specific problem is *cyclic dependencies*

That's where modules depend on each other

In that case, there is no easy order to develop them in

For example, suppose an upgrade is needed which affects all the modules

Then the program is going to be broken until *all* the modules are back in working order - that's too long (see aside: agile development)

Development

Avoiding dependency cycles is ***hugely*** important, making development 'easy' instead of 'nearly impossible'

A development step starts with a working program, and adds a feature, which may affect all the modules

But you can find one module which doesn't depend on anything, fix it up, and test it, even though the other modules are all temporarily broken

Then you can fix up another module, which doesn't depend on anything except the first one, and test it

And so on, until the whole program works again

Getting rid of cycles

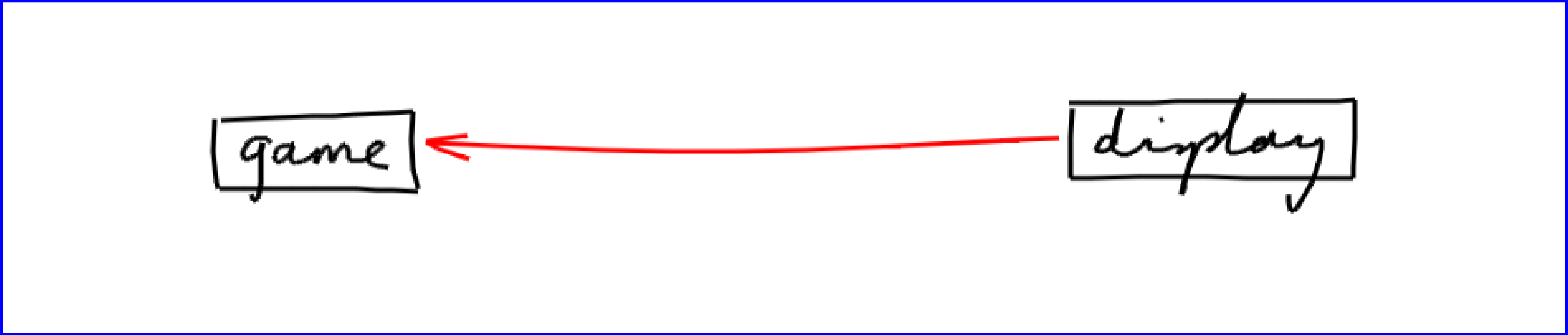
Let's try to get rid of the cyclic dependencies in the sketch

To do that, we have to imagine what the function calls in the implementation are for, and then come up with a better design

Key presses

One dependency that is causing trouble is display depending on game

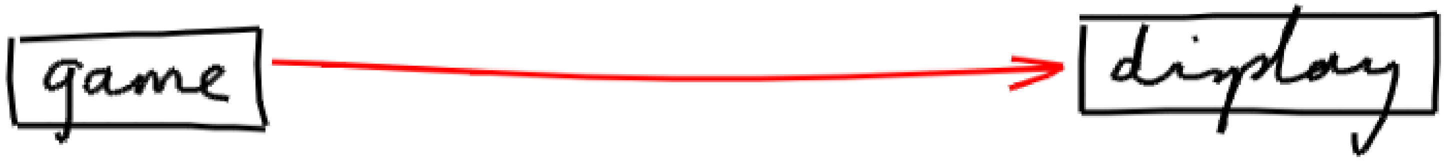
This is presumably because display calls a function in game when a key press is detected



Reversal

A good plan is to reverse the dependency, so that game depends on display

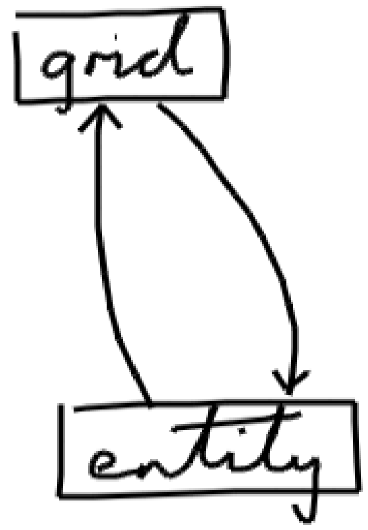
Game can call display to ask for the next key press



The display will need an event queue, but there is usually a need for an event queue anyway

Grid and entities

Another cycle problem, seemingly inevitable, is that the grid needs to know about the entities in it, and entities need to call grid functions to find their neighbours



Grid implementation

A good solution to this is to stop the grid depending on the entity module

Although the grid stores entities, it doesn't need to know anything about the entities, or to call any functions on them

So the grid can be generic, i.e. we can define it to store objects of any type - "grid of anything"

Generic modules

The C language does not provide generic types (except arrays, e.g. `t a[];` means `a` has type "array of `t`")

All C provides is void pointers, of type `void *`

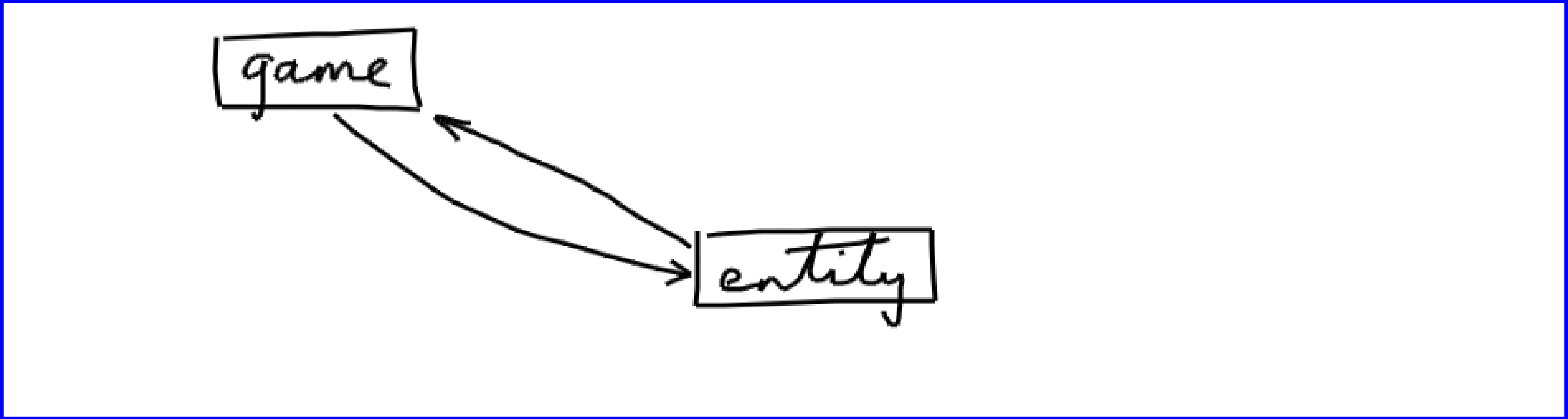
A void pointer variable can hold a pointer of any type

This is "official", so you don't normally need casts, but beware because the result is not properly typesafe

(The lack of safety is not usually a big practical problem, because the kind of bug that it leads to is fairly rare)

Game and entities

One further cycle problem is that the game depends on the entities because it acts as a controller, and the entity module depends on the game because entities need to update the global game state, e.g. the score



Play and state

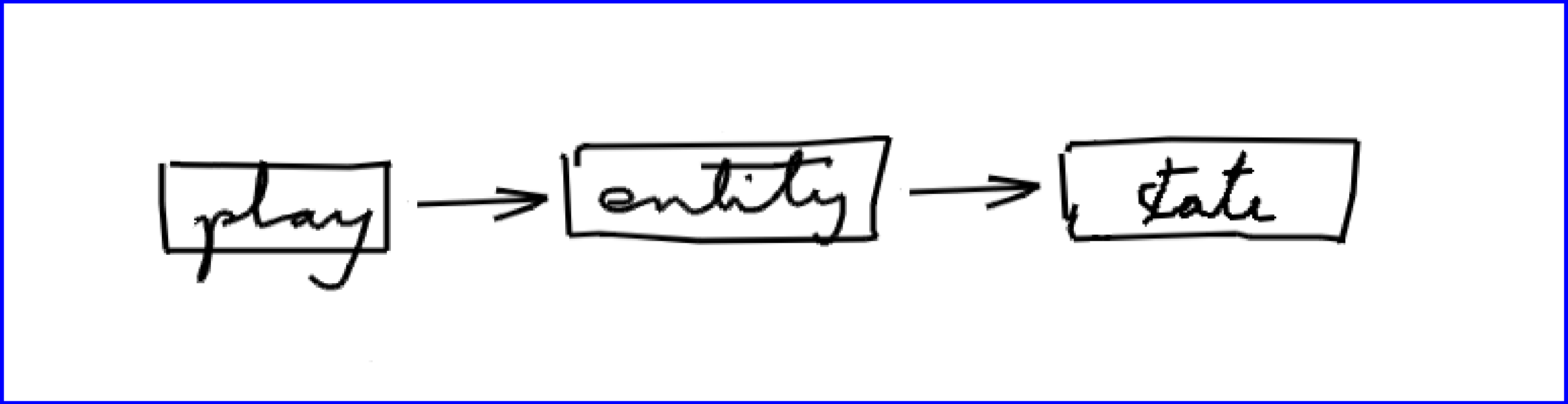
The problem is we have one module which both acts as a controller and keeps track of the global game state

A good solution is to split it into two modules

Let's put the controller aspects into a module called play, and the game state aspects into a module called state

State

We are aiming for this situation:

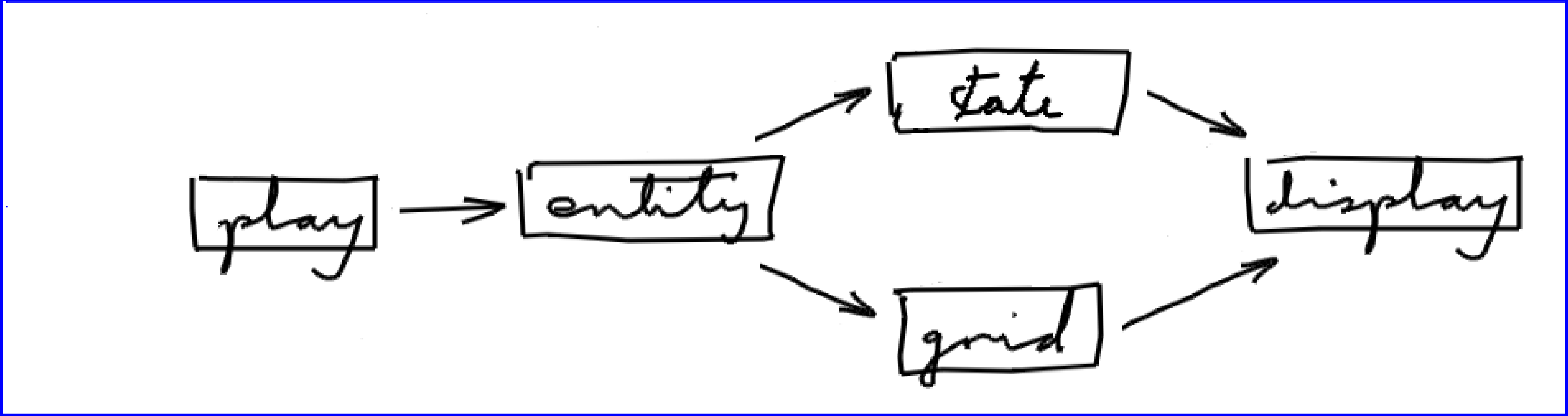


It is possible that the state module might still depend on the entity module, because the state needs to store entities (in our case just the player entity)

But it doesn't need to call entity functions, so we can make it generic again

No cycles

The design changes we've come up with have left us without cycles:

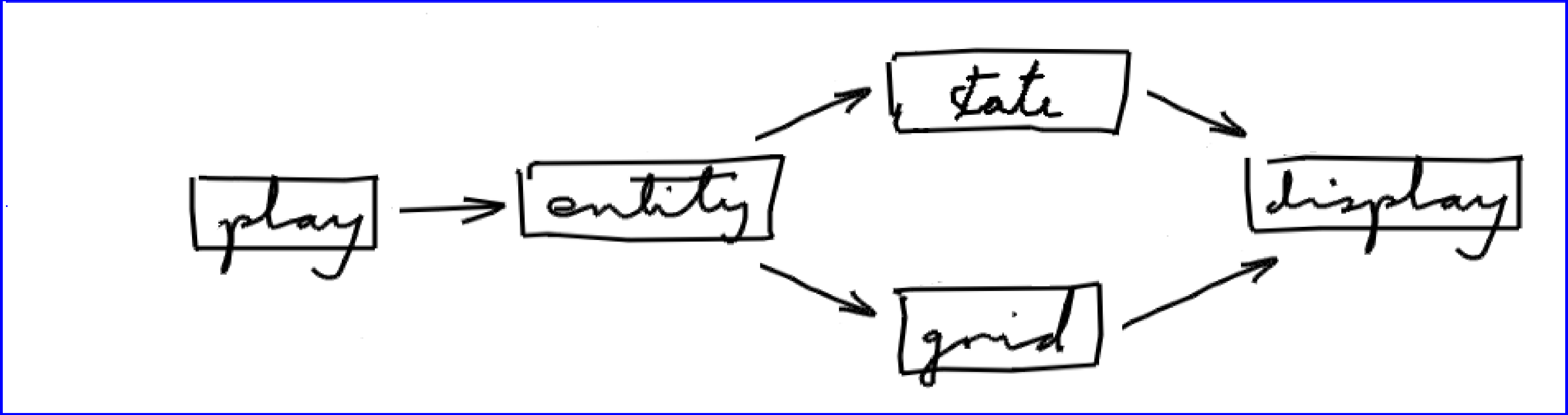


To simplify, indirect dependencies are being left out

If A depends on B and B depends on C, there is no need to draw an arrow from A to C

Shared modules

The display module is a shared one:



Its header file gets included in `state.h` and `grid.h`, and they both get included in `entity.h`, so `display.h` would be included ***twice*** when compiling the entity module

Header problem

It turns out that there is a problem when the header of a shared module gets included twice

Something could get defined twice, which the compiler might not accept

This is particularly a problem with typedefs (which C counts as definitions rather than declarations)

Tree solution

One solution is not to have any shared modules

In that case, the dependency picture is a tree

There are often 'utility' or 'library' modules which get used in many other modules, but if they are only used in the *implementation* of other modules, then their headers only need to be included in `.c` files and all is well

Still, this solution is often not realistic

Simplicity solution

Another solution is to keep header files really simple, containing only structure and function declarations:

```
// Stacks
#include <stdbool.h>

struct stack;

struct stack *newStack();
void push(struct stack *s, int n);
int pop(struct stack *s);
int top(struct stack *s);
bool empty(struct stack *s);
```

Each `.c` file can define its own typedefs

Preprocessor solution

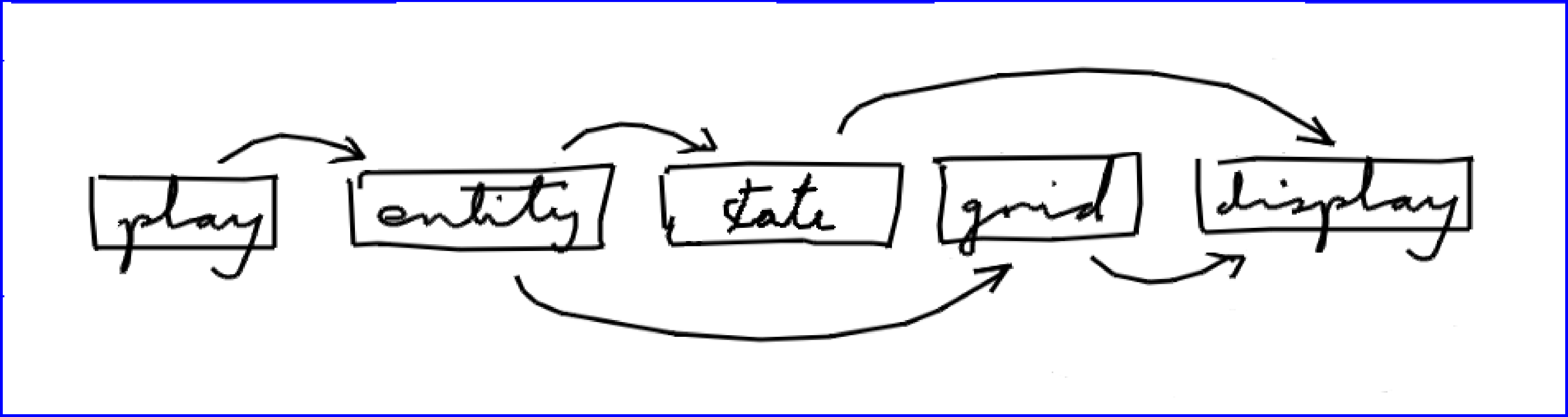
What tutorials usually recommend is to surround a header or an individual type declaration with a protective `#ifdef` guard which prevents it from being included twice

This is used extensively for *library* modules, but it uses preprocessor trickery which some programmers prefer to avoid in ordinary project modules

See wikipedia entry for details

Linear solution

Another approach is to fix on a linear ordering:



The lack of cyclic dependencies makes sure that we can do this, with all arrows going to the right

Then, in each module header, include the header of the module to its right

Graphics problem

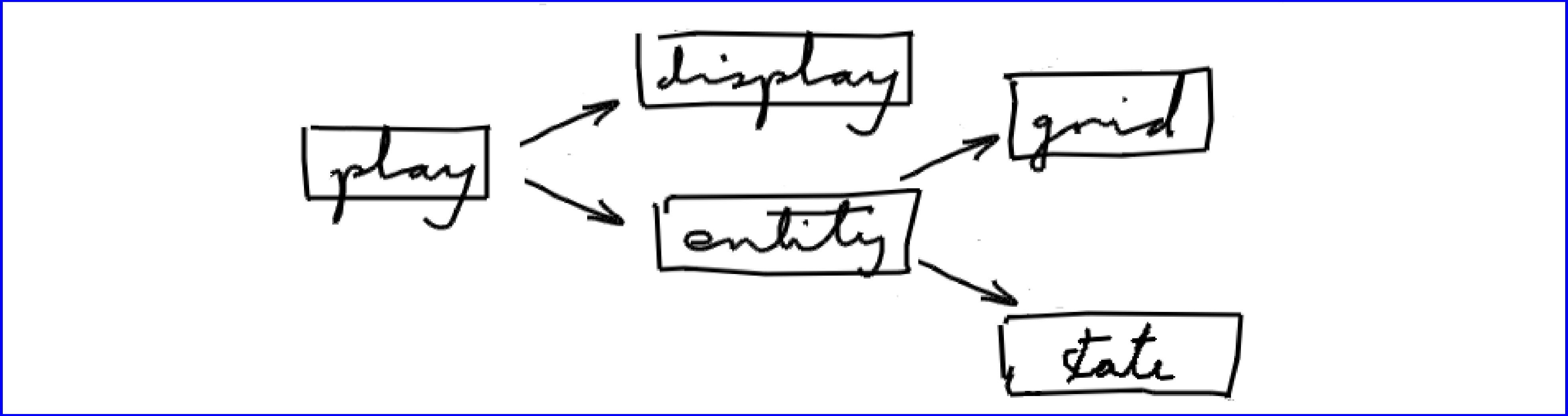
We have another problem anyway - currently, everything depends on the graphics in the display module

It is perfectly possible to design a program this way, and it is very common, but it has a disadvantage

Graphics makes automatic testing difficult, so in our case, graphics makes the auto-testing of ***every*** module difficult

Testable design

A better idea is this:



The three modules that don't depend on graphics represent the logic of the game, and can be auto-tested

The play module drives the logic modules, extracts data from them, and gives it to the display module

Maze program

So let's start development

A good place to start is with a failure function, that you can call to 'crash' the program if you detect a bug

```
void fail(char *message) {
    fprintf(stderr, "%s\n", message);
    exit(1);
}
```

It is only a slight convenience, but it helps to avoid the temptation to leave out defensive safety tests

Base module

Let's put it in a base module, and add general utility functions whenever we like:

```
void fail(char *message);                    base.h
...
```

```
void fail(char *message) {                   base.c
    ...
}
...
```

The comments are in `base.h` not `base.c`

Inclusions

The inclusions in the `.c` file are:

base.c

```
#include "base.h"
#include <stdio.h>
#include <stdlib.h>
```

A module should always include its own header file, so the compiler checks consistency

`stdio/stdlib` are used just for implementation, so can be included just in `base.c`, not `base.h`

"..." means local module, <...> means system module

Breaking the rules

We have ***already*** done something different from our module sketch

And it introduces a shared module

But fortunately, the header `base.h` is simple enough that it can be included multiple times without trouble (the 'simplicity solution')

The rest of the modules won't be shared (the 'tree solution')

Auto-testing

Now is the time to start auto-testing - don't put it off!

We will need multiple `main` functions for testing each module, and a complete program cannot contain more than one `main`

The simplest solution (involving the least trickery) is to have one main program file per module, so for the base module, we will add `baseT.c` (`T` for test), even though there isn't anything that needs to be tested

The test file

The file baseT.c can just be:

baseT.c

```c
// Test the base module.
#include "base.h"

int main() {
    succeed("Base module OK");
}
```

Each module will have three files .h, .c and T.c

Makefile

At the same time, we should start developing a makefile, with one entry to compile each module for testing, e.g.

Makefile

```
.PHONY: base
GCC = gcc -std=c99 -O3 -pedantic -Wall -Werror -o maze
base:
	$(GCC) baseT.c base.c
```

The link shows the full makefile that eventually results

It uses `.PHONY` to say that the targets are not files and compilation should always be done, and defines `GCC` so that the gcc options don't need to be repeated

The program produced is always called `maze`

Grid module

The grid module provides a direction type:

```
enum direction { ... }                               grid.h
typedef enum direction direction;
```

It also has a `nextCell` function to find the entity in a neighbouring cell in a given direction

That way *all* coordinate calculations for finding neighbouring cells are done in one place, in the grid module

Coordinates

There are two common coordinate systems in maths, cartesian coordinates (x right, then y up) and matrix row/column coordinates (r down, then c right)

The most common convention in computer science is graphics coordinates: x right, then y down, a compromise because a display is both a cartesian space and an array of pixels

So that's what we will use for the grid (what matters is to be explicit, consistent and clear)

Grid type 1

There are lots of possibilities for the grid type, e.g. define width and height as constants:

```
enum { width = 9, height = 9 };
struct grid {
    entity *cells[width][height];
};
...
entity *e = g->cells[x][y];
```

Constants

You can define the constants using `#define`

That's fine ***in this case***, but `#define` generally is full of pitfalls and workarounds, so it is worth avoiding

An anonymous `enum` works well for for integer types

Note that `const` doesn't work! (because it means 'read-only', not 'constant')

But in any case, using constants is inflexible (you can't create multiple grids of different sizes, and you can't change the size without recompiling)

Grid type 2

Alternatively, instead of a 2-d array, we could use an array of pointers to column arrays

```
struct grid {
    int width, height;
    entity ***cells;
};
...
entity *e = g->cells[x][y];
```

This uses a lot of pointers, and each column array has to be allocated separately

Grid type 3

We could use a 1-d array, and do our own 2-d indexing:

```
struct grid {
    int width, height;
    entity **cells;
};
...
entity *e = g->cells[x * g->height + y];
```

This still needs separate allocation for the cells array, and the 2-d indexing is not very readable

Grid type 4

We could do the same, but use a C99 feature:

```
struct grid {
    int width, height;
    entity *cells[];
};
...
entity *e = g->cells[x * g->height + y];
```

A structure can have a flexible array as its last element, saving a pointer and allocation call (by allocating extra space for the structure, the array can have any size)

Grid type 5

We could add a pointer trick:

```
struct grid {
    int width, height;
    entity *entities[];
};
...
entity *(*cells)[g->height] = &g->entities;
entity *e = cells[x][y];
```

By defining `cells` as a pointer to a column array, we can use normal 2-d indexing again

The grid module

The grid module uses the 4th approach, on the basis that it is the least unreadable, there aren't many places where 2-d indexing is done, and whatever technique is chosen is not visible outside the module

grid.c

```
struct grid {
    int width, height;
    entity *cells[];
};
...
g->cells[x * g->height + y];
```

Sentinels

Sentinels are sometimes used to avoid edge-case programming

With the grid, we can make sure there are walls all the way round the edges (not necessarily visible on screen)

That way, if a calling function looks for a neighbour in a given direction, this will never go outside the grid

There is still an internal test for out-of-bounds coordinates, as a defensive measure to detect bugs, but callers need not be aware of it

Static functions

The grid module has some static functions in it

```
static int dx(direction d) { ... }          grid.c
```

In this context, the keyword `static` means that the function is not visible outside the `grid.c` file - it is just for internal use

The compiler can do extra optimising (e.g. inlining) knowing that it can see all the calls, and the function can't cause a name clash with other modules

Test file

The grid module has a test file:

```
int main() { ... }                                    gridT.c
```

It does fairly minimal testing, to illustrate how to use the module, and to make sure nothing is broken

How much testing should you do? (a) automate the manual testing you would have done anyway (b) just enough to give yourself or your team confidence

State module

The state module defines the state type, which tracks the player and the number of stars left to find:

state.h

```
struct state;
typedef struct state state;
```

state.c

```
struct state {
    void *player; int stars;
};
```

stateT.c

```
int main() {
...
}
```

Entities

With the entity module, a new issue arises

An entity stores its own (x,y) position in the grid

A problem is keeping its (x,y) fields consistent with its actual location in the grid

It is important to isolate coordinate handling as much as possible, so only a few functions are responsible for this consistency

A good approach is to break the entity module in two

Two modules

The entity module itself will provide just a few primitive but powerful functions which use the coordinates

This will be very stable, and potentially re-usable from one game to another

A new action module will define the behaviour of the different kinds of entity, but it won't have access to the (x,y) coordinates, will only use the functions from the entity module, and ***cannot*** break consistency

Kinds

To specify different kinds of entity, there is a kind type:

```
typedef char kind;                                    entity.h
```

Characters are used as kinds, so that level descriptions can be text-based

Actual constants for kinds of entity are not defined here because the entity module itself is generic

Entity structure

The entity structure is:

```
struct entity {                                              entity.c
    kind k; int x, y; state *s; grid *g;
};
```

Having references to the state and grid objects means that an entity can act autonomously

This is another aspect of object oriented programming - making objects autonomous often helps to improve a program's organisation

Entity functions

The most important functions are:

```
void move(entity *e, entity *target);          entity.h
void mutate(entity *e, kind newKind);
```

The player moves by calling `move`, which swaps the player entity with a blank space next to it in the grid

When a star is collected, `mutate` is called to make it disappear by changing it into a blank space

These can support quite a wide variety of games

Entity testing

As usual, the entity module has a test file:

```
int main() { ... }                                entityT.c
```

The action module

The action module defines the kind constants, and the behaviour of the individual kinds of entity:

action.c

```
enum {
    BLANK='.', WALL='#', STAR='*', PLAYER='@'
};
```

Action functions

The action module defines three functions:

```
void wake(entity *e);                              action.h
void die(entity *e);
void act(entity *e, direction d);
```

The `wake` function is called on each entity at the start, so it can affect the initial game state

`die` is called on an entity (a star in our game) when it disappears, to update the game state

`act` is called on an active entity (the player in our game) to get it to take one step

The wake function

The wake function looks like this:

action.c

```
void wake(entity *e) {
    state *s = getState(e);
    kind k = getKind(e);
    if (k == PLAYER) setPlayer(s, e);
    else if (k == STAR) addStar(s);
}
```

The player records itself in the state

A star adds to the count in the state

Action testing

As usual, the action module has a test file:

```
int main() { ... }                                    actionT.c
```

The display module

The display module defines constants for the keys pressed by the user:

display.h

```
enum key {
    LEFT='<', RIGHT='>', UP='^', DOWN='v', SPACE='.'
};
typedef enum key key;
```

Display functions

It also provides functions:

```
key getKey(display *d);                                display.h
void drawEntity(display *d, int k, int x, int y);
void drawFrame(display *d);
```

`getKey` waits for the user to press an arrow key or space, and returns the key pressed

`drawEntity` draws a single cell into a window image

`drawFrame` transfers the whole window image onto the screen to update what the user sees, and then delays for 20 milliseconds (to support animation)

SDL library

The `display.c` file uses SDL:

```
#include <SDL2/SDL.h>                       display.c
```

The `display.h` header file does not include any SDL headers or mention any SDL functions or types

And no other module includes any SDL headers or mentions any SDL functions or types

Display structure

The display structure is:

```
struct display {                                          display.c
    int width, height, imageWidth, imageHeight;
    SDL_Window *window;
    SDL_Surface *surface;
    SDL_Surface *images[128];
};
```

The SDL window and surface are needed for drawing, and the images for the different kinds of entity are stored so they only get loaded from files once

Error handling

Programs using the SDL library can be difficult to debug, so care is taken to catch SDL errors, according to the SDL function documentation, and report them:

display.c

```
SDL_Surface *image = SDL_LoadBMP(path);
if (image == NULL) SDL_Fail("Bad image file");
```

Drawing

The simplest strategy is used for drawing:

```
SDL_BlitSurface(image, NULL, d->surface, box);      display.c
...
SDL_UpdateWindowSurface(d->window);
```

Each cell is drawn into an image in memory (the 'window surface') and then that image is used to update the screen once per frame

Testing

The display module has a test file:

```
int main() { ... }                                      displayT.c
```

The module can't be automatically tested, so instead it is 'manually' tested – the test function creates a window for a few seconds, to be checked by eye

Warning: you can't run an SDL program from inside a makefile

The play module

The play module is sufficiently small that it doesn't need to be separate

It can be in `maze.c`, so it is the main program

It brings everything together:

```
int main() { ... }                                    maze.c
```

The files

The complete set of files is:

- maze.c
- display.h
- display.c
- displayT.c
- action.h
- action.c
- actionT.c
- entity.h
- entity.c
- entityT.c
- grid.h
- grid.c
- gridT.c
- state.h
- state.c
- stateT.c
- base.h
- base.c
- baseT.c
- Makefile
- images/blank.bmp
- images/player.bmp
- images/star.bmp
- images/wall.bmp