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

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

```
gcc -std=c11 -Wall program.c -o program
```

Multiple modules

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

print0.c

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

main0.c

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

```
gcc -std=c11 -Wall main0.c print0.c -o program
```

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 lot's of places, every time we want to use the module

Header files

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

print.h

```
void print();
```

print.c

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

main.c

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

```
gcc -std=c11 -Wall main.c print.c -o program
```

Own header

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

print.c

```c
#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

Local functions

What if we want a function to be local to a module?

```
// Swap two items in an array
void swap(int array[], int i, int j) { ... }

// Sort an array of integers
void sort(int n, int array[n]) { ... }
```

In a sorting module, we would want to export `sort`, but not `swap` which is a supporting function

Name clashes

Of course, we won't mention `swap` in the header

But that's not enough, because the function name `swap` might clash with something else called `swap` in some other module

The answer is to define `swap` as `static`:

```
static void swap(int array[], int i, int j) ...

void sort(int n, int array[n]) { ... }
```

`static` means local to this module

Extern

The opposite of `static` is `extern`

For functions, it is the default and you don't need it

It is only needed for global variables, which we are not using in this unit

It is used to declare a global variable which is defined in some other module - otherwise you can't tell the difference between a declaration and a definition

#define

Programmers used to use `#define` to make functions efficient:

```
#define twice(n) n+n
#define square(n) n*n
```

But this is mindless textual substitution, so `twice(3)*2` becomes `3+3*2` giving `9` instead of `12`

Brackets

The problem can be lessened by adding brackets round the expression:

```
#define twice(n) (n+n)
#define square(n) (n*n)
```

But this is still mindless substitution, so `square(2+3)` becomes `(2+3*2+3)` giving `11` instead of `25`

More brackets

The problem can be lessened by adding more brackets round the args:

```
#define twice(n) ((n)+(n))
#define square(n) ((n)*(n))
```

But this is still mindless substitution, so if `read` is a function for reading a number then `twice(read())` becomes `read()+read()` which calls `read` twice instead of once

And this time there is no quick and dirty fix

Inline

The right solution in C11 is:

```
inline int twice(n) { return n+n; }
inline int square(n) { return n*n; }
```

This is a hint to the compiler to do inlining (though it might choose not to if the optimisation level is low, and it might do it anyway without the hint if the optimisation level is high)

Inlining is intelligent substitution - it doesn't just save a call, it enables further optimisations on the inlined code

Inline in headers

The compiler compiles each module independently, and links afterwards

So it can only inline a function into its own module (the one containing its definition)

So what if you want to inline a library function?

Old answer: break the usual rule and put the inline ***definition*** of the function into the library header

Cross-module inlining

Putting a function definition in a header is ugly - is there a better way?

Yes, define the function as `extern inline` (`extern` is needed to satisfy `clang`)

```
extern inline int twice(n) { return n+n; }
```

Then use the option `-flto` (link time optimisation)

Array list module

Suppose we implement array lists with

```
struct list {
  int length, capacity;
  int *items;
};
typedef struct list list;
```

And we add some nice functions, and put them in a module

What should the header look like?

Array list header

The list header should be:

list.h

```
struct list;
typedef struct list list;

list *newList();
add(list *ns, int n);
print(list *ns);
```

The declaration `struct list` tells other modules that there is a list structure, but not the fields inside it

It is just like a function declaration (how to access, but no internals)

Opaque types

This:

```
struct list;                                                    list.h
```

is called an ***opaque type*** because you can't see inside it

`list.c` fills in the details:

```
struct list { ... };                                            list.c
list *newList() { ... }
static void expand(list *ns) { ... }
void add(struct list *ns, int n) { ... }
void print(list *ns) { ... }
```

The demo program

The program that uses `list.h` is:

listdemo.c

```
#include "list.h"

int main() {
    list *numbers;
    numbers = newList();
    add(numbers, 3);
    add(numbers, 5);
    add(numbers, 42);
    print(numbers);
}
```

Encapsulation

The list module as a whole has been ***encapsulated*** (given a protective wrapper)

From `listdemo.c` you cannot misuse the list module, you can only call the functions provided

You can't even call `malloc(sizeof(struct list))` because the compiler doesn't know the size

This is a very robust way to write multi-module programs

Organisation

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

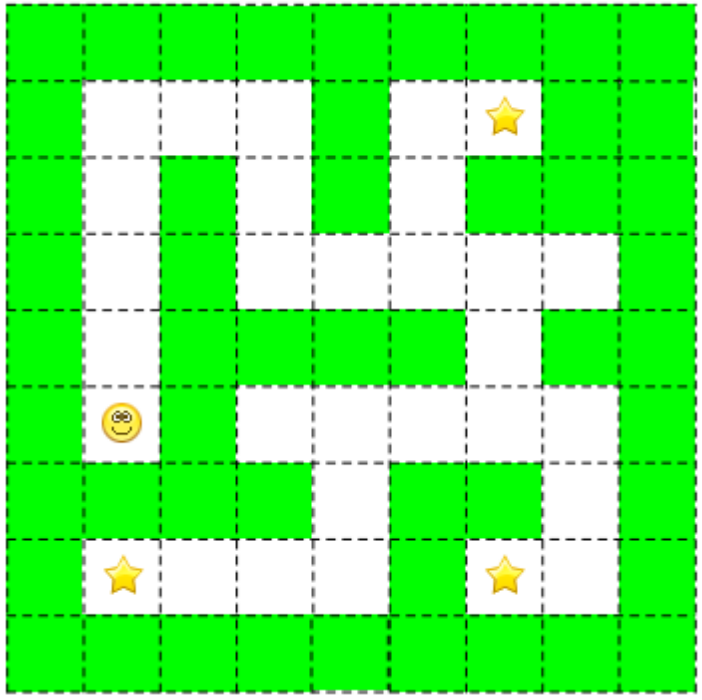
What's different is the organisation of the functions

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

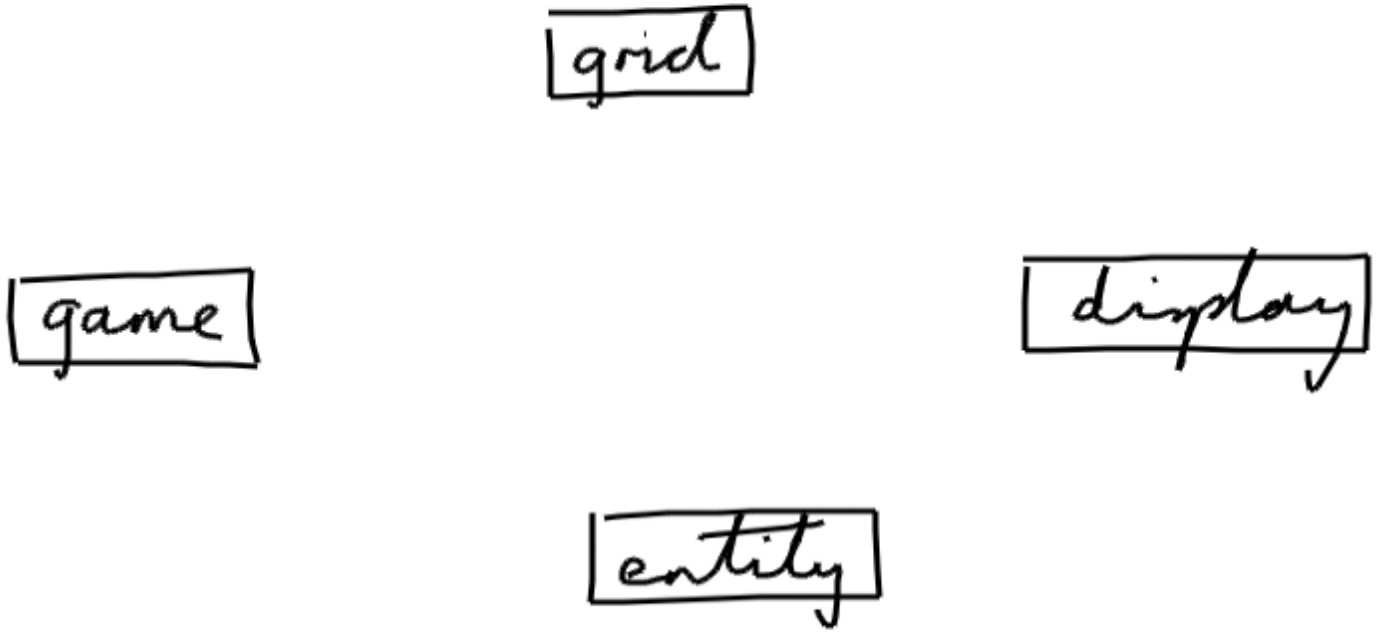
Sketch

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

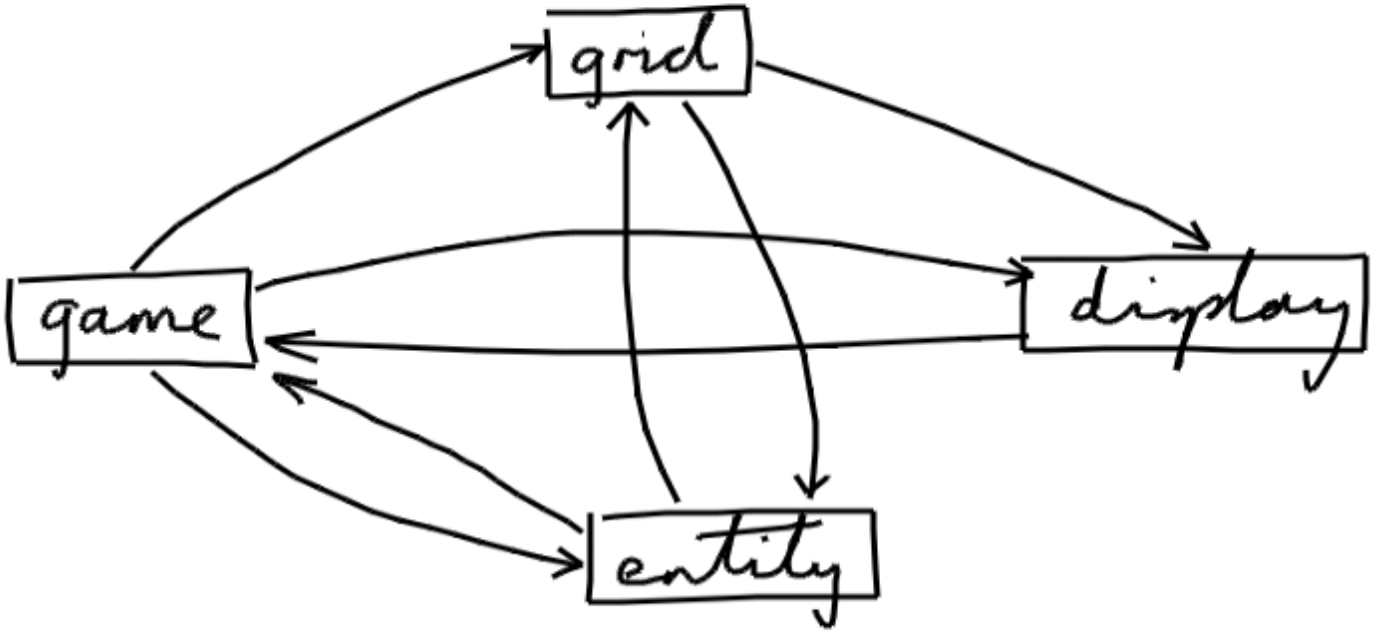
A module A *depends on* a module B if a function in module A calls a function in module B

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

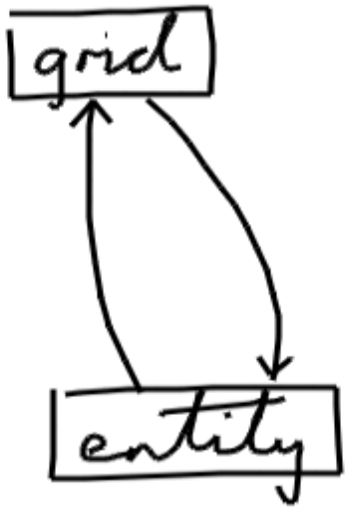
A good plan is to reverse the dependency, so that game depends on display

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)

Forward references

Maybe having a generic grid module based on void pointers seems too complex, or too unsafe

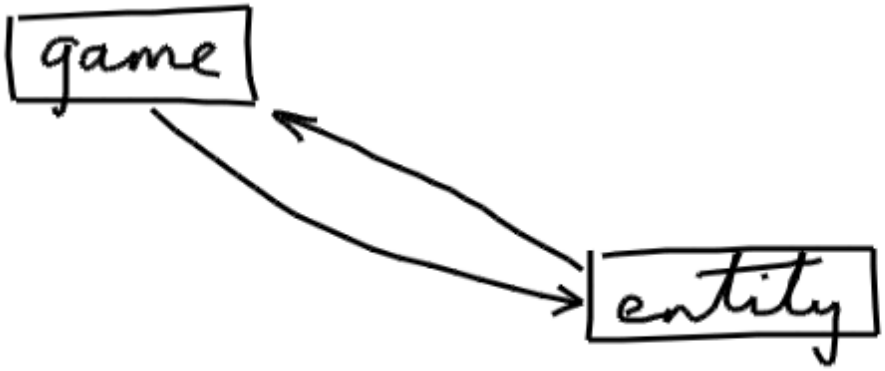
An alternative is a forward reference

The opaque type `struct entity` can be declared in the grid header `grid.h` instead of the entity header `entity.h`

The grid module knows nothing about it, so can only handle it via pointers, and its details get filled in by the entity module `entity.c` as before

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

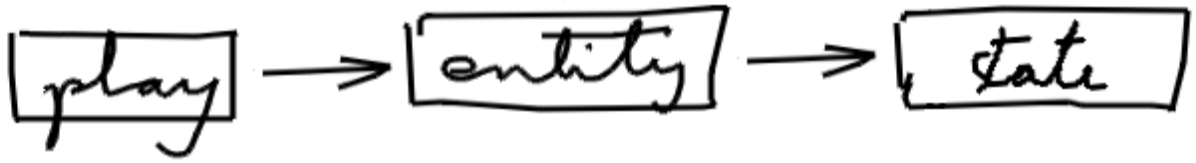
The problem is we have one module which both acts as a controller and keeps track of the global game state

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:

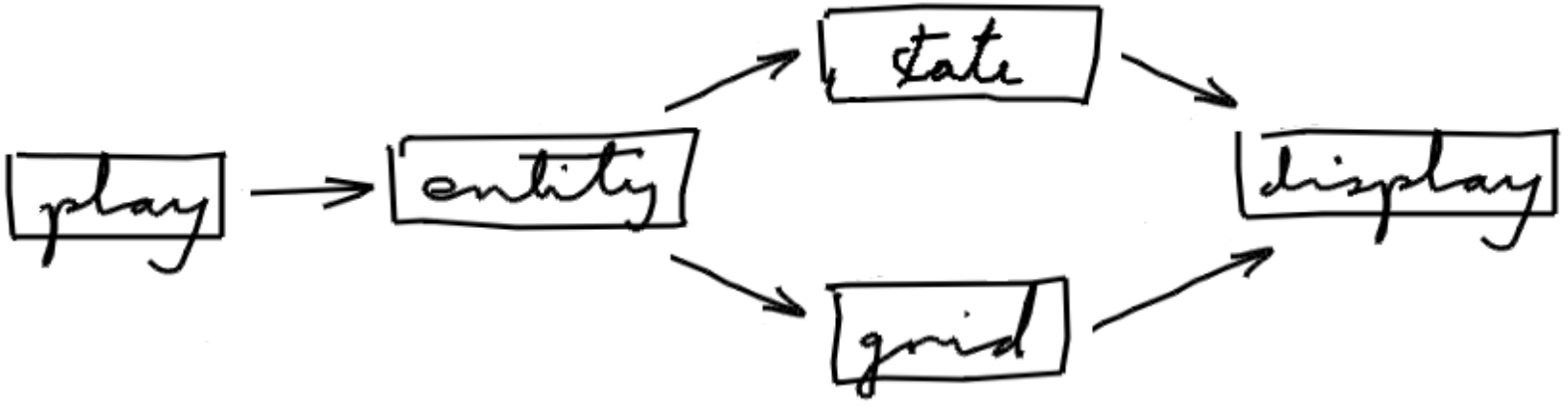


It is possible that the state module might still depend on the entity module, because the state needs to store entities (in our case just the player entity)

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

No cycles

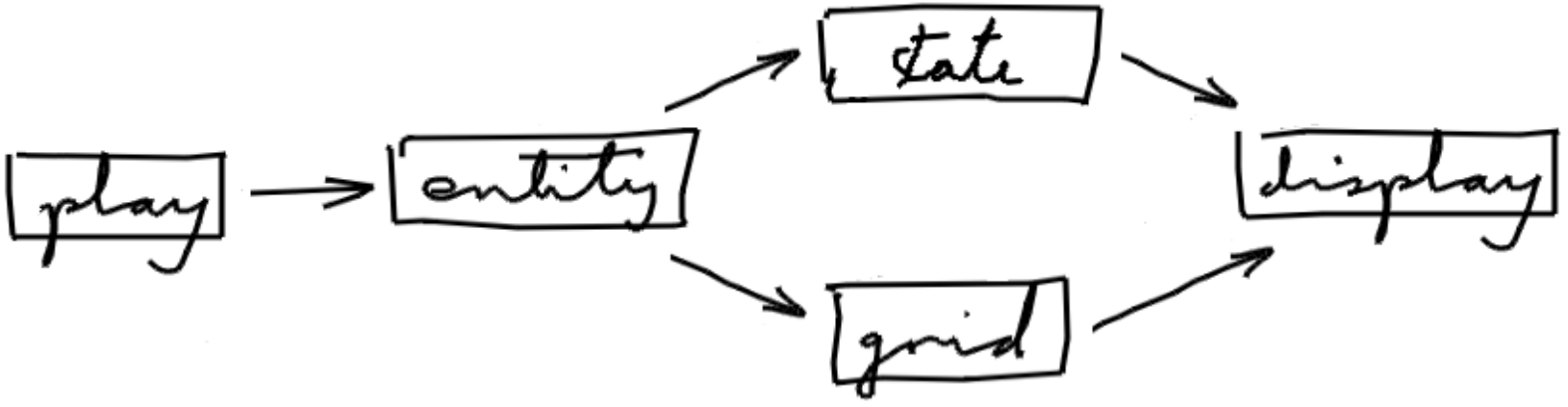
The design changes we've come up with have left us without cycles:



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

Typedefs

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

In C99, there used to be a problem with typedefs (which C counted as definitions rather than declarations), but the typedef problem has ***gone away*** in C11 (the same typedef can appear multiple times)

Enumerations

That leaves enumerations as the only remaining problem - if a header contains:

```
enum suit { Club, Diamond, Heart, Spade };
typedef enum suit suit;
```

then that header can't be included twice

Solution 1

One possibility is to define the constants as `const` variables:

```
extern int const Club, Diamond, Heart, Spade;
typedef int suit;
```

The `extern` means "these variables are defined somewhere in the program, but not here" so this declaration can be repeated without trouble. The actual definitions are in a `.c` file.

The type `suit` now has to be `int`. The constants can't be used in switch statements.

Solution 2

Or: define the enumeration in its own module, with functions but with no other custom types, and define `suit` as a synonym for `int` rather than `enum suit`:

```
enum suit { Club, Diamond, Heart, Spade }; suit.h
typedef int suit;
```

In any other header which needs the `suit` type, write `typedef int suit;` not `#include <suit.h>`.

In a `.c` file that needs the functions, use `#include <suit.h>`.

Preprocessor solution

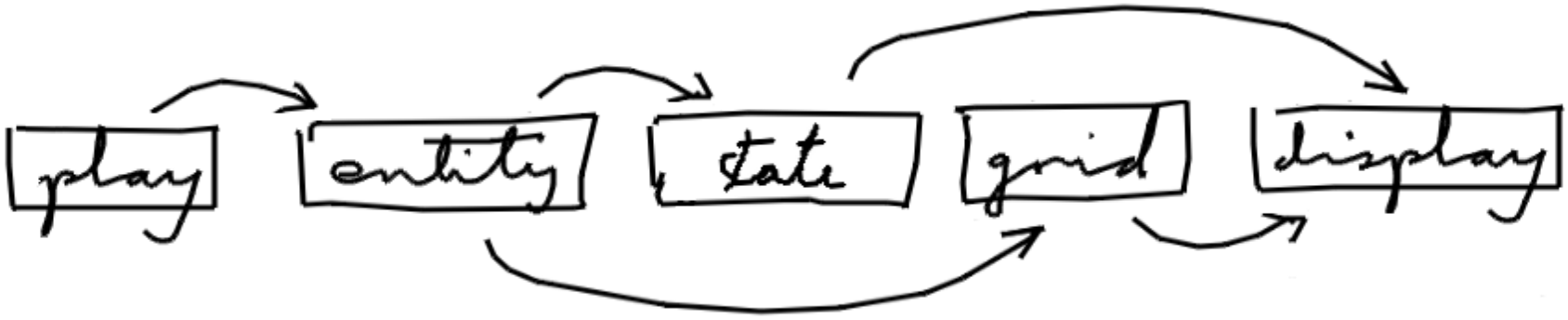
What tutorials usually recommend is to surround a header or an individual type declaration with a protective `#ifdef` guard which prevents it from being included twice

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

See wikipedia entry for details

Linear order

With no cycles, modules can be put in a linear order:



Then any development step can upgrade and test the modules one-by-one from right to left

Graphics problem

We have one problem left - 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

Model-View-Controller

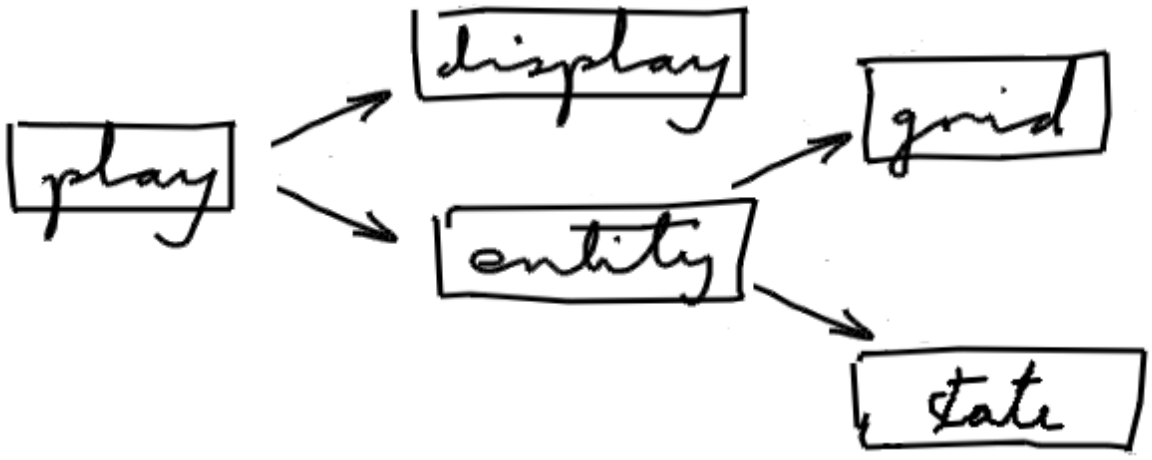
You often hear about the "MVC design pattern" for programs with (graphical) user interfaces

The problem is, there is no consensus about how the three groups of modules depend on each other

A good approach is to have the controller depend on the other two, which are independent of each other

Final design

That leads to this final design:



The entity/grid/state modules form the "model" (the logic), and can be auto-tested, the display module is the view, and the play module is the controller

The play module drives the model, extracts data from it, and gives it to the view

Maze program

So let's start development

It will very soon turn out that nearly every module needs a `fail` function to print a message and exit

Defining it once and for all helps to avoid the temptation to leave out safety tests

So let's create a `base` module to hold `fail` (and `succeed` as a very minor convenience)

Change of design

So ***at the very first step***, we need to change our design by adding a module – that's totally normal

base.h

```
/* The base module provides success and fail
functions. */

// Print with newline, avoiding the need for stdio.h.
void succeed(char *message);

// Report a test failure or bug, and stop the program.
void fail(char *message);
```

Auto-testing

Now is the time to design 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` - let's use conditional compilation:

base.c

```
#ifdef baseTest

int main() { ... }

#endif
```

For testing the module, `baseTest` is defined and `main` is included, otherwise `main` is left out

Makefile

At the same time, we should start developing a makefile:

Makefile

```
base = base.c

CC = clang -D$@Test -std=c11 -Wall -pedantic -g -o maze

%: %.c
	$(CC) $($@)
	./maze
```

Module list


```
base = base.c
...
```

The variable `base` holds the list of module source files to be compiled when testing the base module.

The idea is that the Makefile begins with the list of modules. It is convenient to make this the authoritative place where they are listed, with dependencies. There is no chance of the list becoming out of date, because it is an active part of the Makefile.

Compiling


```
...
CC = clang -D$@Test -std=c11 -Wall -pedantic -g -o maze
...
```

The variable `CC` is the compile command to use.

In the option `-D$@Test`, the `$@` will be replaced by the name of the module. When you type `make base`, the option becomes `-DbaseTest` which causes the `main` in `base.c` to be included.

All modules are compiled to a `maze` file, to avoid lots of excutable files.

Rule


```
...
%: %.c
        $(CC) $($@)
        ./maze
```

When you type `make base`, `make` looks for a file `base` and doesn't find it (because `maze` is the only executable). But it finds `base.c`, so this rule matches. Then `$@` expands to `base`, so `$($@)` becomes `$(base)` which expands to the relevant list of source files. The module is compiled and then its tests are run by executing `maze`.

What's next

It seems the grid module is next

We want to be able to move round using directions rather than coordinates, so that all coordinate calculations stay inside the grid module

We need an enumerated type to define the directions

According to our conventions, the enumerated type should go in its own module

So, another ***change of design***, we'll add a direction 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 a matrix of pixels

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

Direction header

Here's a suitable header for the direction module:

direction.h

```
/* Directions allow entities to move around in the grid
without using absolute coordinates. Graphics coordinates
are used, i.e. (x,y) with x to the right and y down. Don't
include this header in other headers, just repeat the typedef. */

// There are eight directions, plus Here for the current position.
enum direction {
    Here, North, South, East, West,
    NorthEast, NorthWest, SouthEast, SouthWest
};
typedef int direction;

// Find the x offset corresponding to a given direction.
int deltaX(direction d);

// Find the y offset corresponding to a given direction.
int deltaY(direction d);
```

Inclusions

The inclusions in the `direction.c` file are:

direction.c

```
#include "direction.h"
#include "base.h"
#include <stdbool.h>
#include <assert.h>
```

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

`stdio/stdbool` are used just for implementation, so can be included just in the `.c` file, not the header

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

Makefile

The Makefile needs to be updated:

Makefile

```
base = base.c
direction = direction.c base.c
...
```

This lists the direction module, and its dependency on the base module. The modules are listed in reverse dependency order, which is the order of development or refactoring. Typing `make direction` gives both `direction.c` and `base.c` to the compiler, but only the testing from `direction.c` is included.

Grid module

The grid module 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

The grid module has a forward reference to the entity type, and is written in a generic style without using any entity functions, to avoid a dependency on the entity module

Grid type 1

There are lots of possibilities for the grid type

The simplest is to define width and height as constants:

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

The problem is, the size can't be changed without recompiling

Grid type 2

The most straightforward flexible approach is to use an array of pointers to column arrays

grid.c

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

This uses a lot of pointers, and each column array is allocated separately - this is what our grid module uses

And there are a lot of intermediate possibilities

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

Makefile

The Makefile needs to be updated again:

Makefile

```
base = base.c
direction = direction.c base.c
grid = grid.c $(direction)
...
```

This lists the grid module, and its dependencies on the direction and base modules. Using the variable `$(direction)` is just an abbreviation for two files. Typing `make grid` compiles and tests the grid module.

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 {
    entity *player; int stars;
};
...
```

It is 'generic', and the header has a forward reference to the entity type.

Makefile

Another entry is needed in the Makefile:

Makefile

```
base = base.c
direction = direction.c base.c
grid = grid.c $(direction)
state = state.c base.c
...
```

This time, the state module doesn't depend on all the previous modules, only on the base module.

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 in the entity module, because the entity module is 'generic'; the `kind` typedef is a forward reference to the action module

Entity structure

The entity structure is:

entity.c

```
struct entity {
    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:

entity.h

```
void move(entity *e, entity *target);
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

Makefile

Another entry is needed in the Makefile:

Makefile

```
base = base.c
direction = direction.c base.c
grid = grid.c $(direction)
state = state.c base.c
entity = entity.c state.c $(grid)
...
```

The variable `$(grid)` is a useful abbreviation for three files. On the other hand, file `state.c` is used instead of `$(state)`, otherwise the base module `base.c` would get included twice.

Action module

The action module defines specific kinds of entity as an enumerated type.

action.h

```
enum {
    Blank='.', Wall='#', Star='*', Player='@'
};
typedef char kind;
```

Since the action module does not export any other types, there is no header inclusion problem, provided other headers repeat the typedef instead of including the header.

Action functions

The action module defines three functions:

action.h

```
void wake(entity *e);
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

Makefile

Another entry is needed in the Makefile:

Makefile

```
base = base.c
direction = direction.c base.c
grid = grid.c $(direction)
state = state.c base.c
entity = entity.c state.c $(grid)
action = action.c $(entity)
...
```

The display module

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

A separate module to define an enumerated type is avoided by defining the keys as external constants (though they can't then be used in switch statements)

display.h

```
typedef char key;
extern const key Left, Right, Up, Down;
```

Display functions

The display module 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:

`display.c`

```
#define SDL_MAIN_HANDLED
#include <SDL2/SDL.h>
```

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:

display.c

```
struct display {
    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 =
    P(SDL_LoadBMP(path));
```

A couple of custom functions `P` and `I` help to make error handling look minimal

Drawing

The simplest strategy is used for drawing:

display.c

```
SDL_BlitSurface(image, NULL, d->surface, box);
...
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 function:

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

Makefile

Another entry is needed in the Makefile:

Makefile

```
base = base.c
direction = direction.c base.c
grid = grid.c $(direction)
state = state.c base.c
entity = entity.c state.c $(grid)
action = action.c $(entity)
display = display.c -lSDL2
...
```

The display module depends only on the SDL library, which it needs to be linked with.

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:

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

maze.c

The files

The complete set of files is:

- Makefile
- display.h
- kind.h
- entity.h
- grid.h
- state.h
- direction.h
- base.h
- maze.c
- display.c
- kind.c
- entity.c
- grid.c
- state.c
- direction.c
- base.c
- images/blank.bmp
- images/player.bmp
- images/star.bmp
- images/wall.bmp

Wanderer

The structure of the maze program is suitable for much more complex grid games, e.g. wanderer:

wanderer

It's a reconstruction of a retro game from the 1980's

It has 60 difficult levels, so don't get addicted unless you have time