Hands-On Object-Oriented Programming in C++ Warm up: course prerequisites

1 What is OOP?

Object-oriented programming (OOP) is fundamentally about managing complexity by creating abstractions. As the programs we write grow and become more complex, they become more difficult to write, understand and maintain. The key insight of object-oriented programming is to think about the components of the program as higher-level 'objects' whose internal details do not matter, when viewed from the outside.

It is perfectly possible to write object-oriented programs in languages which make no claim to supporting the OO style explicitly. Languages which do make such claims, may make the process easier by providing built-in abstractions of mechanisms one would typically use when programming in such a style.

During the OOP boom, it became common practice to increase the complexity of programs (unnecessarily) for the sake of use of advanced OOP techniques. This had the effect of making programs more complex and more difficult to manage: exactly the opposite of the desired effect. There is much unnecessary cruft in the OO literature.

In this course we will look at some semi-advanced OO techniques, in order to give you a feeling of how such techniques work, and to give you some experience with a non-trivial C++ code base. These techniques may appear complicated, especially at first glance, but it is important to understand that the long-term goal is to make the program simpler to manage.

Getting a good feeling of when some non-trivial OO technique will provide long-term benefits to the code, rather than unnecessarily complicating it, is something that comes with experience; it cannot be taught in a few days.

If you forget almost everything we learn about OOP in this course, and remember just one thing, please let it be this:

> The essence of object-orineted programming is the use of high-level interfaces to maintain ignorance of the low-level details behind those interfaces.

2 Roadmap

While most features of C++ may seem quite sensible and easy to manage in isolation, complications frequently arise when different C++ language features collide in real programs. Teaching C++ features through the use of trvial examples which show off the features in isolation (thereby avoiding the distractions caused by the large-program considerations) is a vital technique for teaching beginners.

As this is a second-level course, you will be exposed to non-trival code which will give you some appreciation of the difficulties that arise in writing real C++ programs, and, hopefully, give you some experience in resolving such complications.

For the rest of the course, we will be developing a single program. You should record your progress in a Mercurial repository.

While the model itself is rather simple, it does provide a fairly rich playground for exploring interactions between objects and for seeing how decomposition into objects affects our ability to manage complexity.

The downside is that, at times, the complexity inherent to C++ may overwhelm us. At such times it is important to take a step back, and look at the high-level picture as drawn by objects in our program and their relationships.

I will be demonstrating the development of the answers to all the exercises in the course on the projector, after having given you some time to attempt it yourselves. If you get stuck, ask me to help you. If you are too stuck, you may steal my solution.

The evolution of my solution will be available to you throughout the course from my Mercurial repository.

Please beware that, at times, the many low-level details that C++ forces us to think about may seem overwhelming. When this happens it is very important not to get too frustrated and distracted by them: Sit back, relax, steal my code, and make sure you understand the high-level structure of the code the exercise is addressing.

The high-level considerations are the protagonists of this course. Any low-level details you pick up, should be considered as a bonus.

3 Simple classes

Make sure that you can compile and run this code.

Try to get a broad understanding of how the code works. You do **not** need to understand any of the OpenGL-specific code in any detail.

As you can see, the code is not particularly well organised. Improve it by creating an abstraction for the object which moves around the screen:

Write a `Particle` class.

Make sure to commit your changes to the repository, once you are happy that your code works.

Don't try to do too much at once. I recommend starting with a class which has 5 `float` data members, and two member functions. The bodies of the latter you can cut and paste from the code you have been given.

3.1 Compiler-generated members

Do you need to write a copy constructor for `Particle`?

Do you need to write an assignment operator for `Particle`?

Do you need to write a destructor for `Particle`?

3.2 Container

Introduce a few more particles into the simulation. Store them in an `std::vector`.

3.3 More abstraction

We have tucked the details of particle representation inside a class, but the details of how these particles are drawn and how they are managed are still spread all around the code for all to see.

Encapsulate the details which are spread around the top-level of the code in a `Simulation` class. It should

- hide the container of particles behind an interface which permits adding new particles to the simulation
- hide the OpneGL initalization behind a simple interface which allows the setting of the screen size.

Hints follow.

3.3.1 Reference members

When you try to move the following chunk into the class

```
// A place to store the output of glGetIntegerv
int viewport[4];
// Meaningful synonyms for some of the above
int& x_min = viewport[0];
int& y_min = viewport[1];
int& x_max = viewport[2];
int& y_max = viewport[3];
```

you will need to adjust the initalization style. References must be initialized at the very first (and only) opportunity. For non-members the first opportunity is at the point of declaration. For members the first opportunity is in the initializer lists of constructors.

```
class Simulation {
  Simulation(/* ... */)
   :  x_min(viewport[0]) /* ... */ {}
  // ...
};
```

Note: eventually you will be forced to rethink this idea.

3.3.2 Static member functions

Passing `Simulation::display()` as the callback function to `glutDisplayFunc` does not work. Remember that member functions have a `this` pointer as their hidden first parameter: `glutDisplayFunc` expects a function which has no parameters at all.

One way to get around this would be to revert to having `display` as a global function. Another way would be to keep `display` as a member function of `Simulation` but suppress its `this` pointer. The way to do this is to make it a `static` member function:

```
class Simulation {
  // ...
  static void display();
  // ...
};

void Simulation::display() {
  // ...
}
```

Static member functions may be called just like ordinary member functions, through (explicit or implicit) `this`

```
void member_function() {
   this->static_member_function();  // explicit this
   static_member_function();        // implicit this
}
```

or via the class, with the scope resolution operator:

```
Class::static_member_function()
```

Once you remove `this` from `Simulation::display`, the latter will no longer be able to acces the (non-static) members of the class. You need a `this` pointer to inform the function which instance of the class it is working on. Without `this` it is not working on any specific instance, so you will need to inform it which instance to use explicitly.

There are now two ways forward.

- Make all members static
- Make the class a *Singleton* and instruct the static functions to use the single instance.

3.3.3 Singleton

Arguably there should only be a single instance of `Simulation` in our program as it would make no sense to run two simulations simultaneously in the same process. The fact that `glutMainLoop()` must be called only once in a single process, is another hint in this direction.

Beware of this design pattern: Firstly, it is the simplest of the famous design patterns and is therefore the only one that many people understand. This leads to it being overused. Secondly, it's just a fancy way of having global state, which is usally better avoided.

Having said that, in this case we can tolerate it. Turn `Simulation` into a singleton as follows

- Make the constructors private, to prevent clients from making new instances of the class
- Create a private static pointer to `Simulation`.
- Provide a static public function with a name such as `get_instance`, which returns the above pointer (or the instance by reference), initializing it with a fresh instance, if necessary.

Note that you will have to make some other members static, if the program is to work.

Now that clients do not call the constructor directly, how will they specify initialization parameters?

4 Separate compilation

Split the code into five separate files:

- `main.cc`
- `Particle.hh`
- `Particle.cc`
- `Simulation.hh`
- `Simulation.cc`

4.1 Header guards

Don't forget to use header guards to protect against multiple inclusion:

```
#ifndef Classname_hh
#define Classname_hh

// .. contents of header file

#endif
```

4.2 Makefiles

At this point we can get away with recompiling all our code every time something changes:

```
g++ main.cc Particle.cc Simulation.cc   # + necessary flags
```

In large projects this would be unnecessarily wasteful: you should only recompile those portions of your code which have been affected by the changes you have made. Keeping track of which portions of the code need to be recompiled, is exactly the sort of task at which computers are much better than humans.

Most IDEs will provide a built-in mechanism for ensuring that you compile no more than necessary. A widely-used and IDE-independent tool is `make` and the associated *makefiles*.

As such mechanisms in general, and makefiles in particular, are neither C++-specific nor are they mentioned at all in the C++ standard, we will not cover them in any depth in this course.

You are provided with the following sample makefiles. You may use them as a starting point in your own projects.

- Makefile_minimal: minimalistic, shows essential components without any explanation.

- Makefile: as above, but with basic instructions for use. Probably the best place to start.
- Makefile_explicit: more sophisticated, with copious comments explaining how it works.

From now on, you should use `make` to compile the code we are writing in this course: copy `Makefile` into your source directory, and type `make`. You may need to adjust some of the flags which are set in the makefile. Don't forget to track the makefile in your version control system.

5 More simple classes

5.1 Colour

Currently, all our particles are red. Make it possible to have particles be drawn in different colours.

One way to do this would be to store three numbers in each particle instance, representing the red, green and blue componens. An alternative approach would be to create a class for representing colours, and to store an instance of that class in the particle.

> Write a `Colour` class and use it in `Particle`.

5.1.1 Varied construction

There are diffeent ways of specifying colours:

- RGB values in the range 0-1
- RGB values in the range 0-255
- RGB values in the range 0-f
- RGB values in the range 0-ff
- CMYK in various ranges
- etc.

Which one of these schemes should the constructor of `Colour` use?

One way of addressing this problem is to provide a collection of pseudo-constructors: static member functions with names which explicitly state which scheme they use (that way the client is not tempted to guess). These pseudo-constructors translate their input into whichever scheme is accepted by the constructor, and use the latter to create the instance.

OK, but which scheme should the *real* constructor accept? Perhaps it is best to force the clients to use the

pseudo-constructors (by making the real constructor private), that way you avoid making the wrong choice in the interface, and reduce the chance of clients guessing the wrong meaning of the constructor's parameters.

> Make the construction interface of `Colour` consist of pseudo-constructors with names such as `rgb1`, `rbg255` and `rgbFF`.

Having just two pseudo-constructors is enough to illustrate the point: I suggest `rgb1` and `rgb255` as these will have the simplest implementations. Only do `rgbFF` if you are waiting for others to catch up.

5.1.2 Private data

How should clients access the colour components?

We could choose to store the data in many different formats (see *Varied constructon* above). By making the actual data we store private, and giving access to them via public functions, we give ourselves the vital ability to

- change our mind about how we store data inside the class,
- keep the client code working in spite of any internal changes we make.

This is a fundamental principle in C++ programming:

> All data members should be private.
>
> If you want them to be public, make them private anyway, and grant access to them via public getter and/or setter functions.

In extremely rare cases, you may, justifiably, violate this principle. We will see a situation where this might be a reasonable thing to do, shortly.

Provide getters such as `r1()` (for reading the red component expressed as a `float` in the range 0-1) and `r255()` (for reading the red component expressed as an integer in the range 0-255). Make the actual data members accessed by these getters, private.

Notice that, if you choose to change the internal representation of the colour, you can adjust the definitions of the getters and pseudo-constructors accordingly, and all client code will continue to work as before: it will not be affected by your change.

1. Getters, setters and performance

   Trivial getters and setters (member functions which merely read or overwrite the value of a member datum) are prime candidates for inlining: having the compiler replace the function call with the function body.

   If you proide the definition of your trivial getters and setters inside the class block, then the compiler will almost certainly replace getter and setter calls with direct reads or writes of the member in question. There will be no performance degradation because of the use of trivial getters and setters.

5.1.3 Convenience symbols

Some colours are going to be used very frequently. It would be more convenient if users were allowed to say `Colour::RED` rather than `Colour::rgb1(1,0,0)`.

Provide convenience symbols for red, green, blue, yellow, black and white.

5.1.4 `const` members vs. assignment operators.

Colours should be immutable: the colour red should always be red, it should not be possible to mutate into some other colour. The standard way of expressing immutability is to use `const`. It seems reasonable to `const`-declare the data members of `Colour`.

Try this, and you will see that the compiler complains about the assignment operator: `Colour& Colour::operator=(const Colour&)`. (Whether this actually happens, depends on a number of choices that you made in the implementation of the program so far. If it isn't happening for you, ask me to show you how to modify your code to make it appear.)

Do you understand why the compiler complains about the assignement operator, in the context of `const`

member data?

Mouse over to see answer.

Is it, therefore, not possible to have an assignment operator for a class which has `const` data members?

Mouse over to see answer.

But that doesn't make sense in our case: If we use the assignment operator to replace some colour, the new colour should take place of the old. So what is the solution in our case?

Mouse over to see answer.

But then we've lost immutability, haven't we?

Mouse over to see answer.

5.2 Vector

The `x` and `y` components of the particles position and speed could benefit from being packed up together in a single object, and being given higher-level operations for their manipulation.

> Write a (2-dimensinal) `Vector` class, and use it to store the positon vector and the velocity vector of `Particle`.

Here is an example of a class where it might just about be acceptable to make member data (the `x` and `y`-components) public. The argument would be that for such a simple class, in which these attributes are such a fundamental and obvious parts of not only the interface but also the implementation, the benefit gained from being allowed to write

```
v.x = 2;
v.x;
```

as opposed to

```
v.set_x(2);
v.get_x();
```

or even

```
v.x(2);
v.x();
```

outweighs the risks involved.

Let's take the risk, and see whether it comes back to haunt us.

5.2.1 Operators

One advantage of having the `Vector` class is that code which was previously written like this

```
position_x += velocity_x * dt;
position_y += velocity_y * dt;
```

can be simplified to

```
position += velocity * dt
```

Notice that if, for example, we were to turn our simulation into a three dimensional one, the former code would have to be changed to include a line updating the `z`-component, and similar changes would have to be made in *every single place* that expresses any operations on vectors.

With the `Vector` class in place, the transition from 2 to 3 dimensions would be much simpler: the implementation of the `Vector` class would have to be modified, but (almost) all the client code would work without any modification at all. Magic!

Notice that in the last code sample operators with the following approximate signatures are used

- `operator * (Vector, float)`
- `operator +=(Vector, Vector)`

I recommend that you first implement `+=`, and then use it in the implementation of `+`. Use a similar trick for `*=` and `*`. In general, you should use *compound assignment* operators in the implementation of their corresponding binary operators.

> Implement operators `*`, `*=`, `+` and `+=` for the `Vector` class, and use them in the client code.

1. What should assignment operators return?

   We are now venturing into areas which are beyond the scope of this course. For completeness here is a brief summary of the answer and its motivation. The fundamental types support both of the following uses:

   ```
   a = b = c;   // a and b take the value of c
   (a = b) = c; // Beyond scope of the course
   ```

   In brief, you should strive to provide analagous behaviour for your types. Therefore

   > assignment operators (including the compound ones) should return *non-*`const` *references* to `*this`.

2. Check for self-assignment

   Consider what happens in the following situation.

   ```
   MyClass m;
   // ...
   m = m;
   ```

   Yes, you are unlikely to write code like this, but it is quite possible to write code which gives rise to the same situation by more circuitous means.

   Specifically, think about what would happen if `MyClass` contains some dynamically allocated memory.

Mouse over to see answer.

The moral of the story is, check for self-assignment in the assignment operator, and do nothing (beyond returning `*this`) when self-assigning:

```
MyClass& MyClass::operator = (const MyClass& other) {
  if (this != &other) {
    // ... deallocate, allocate, copy
  }
  return *this;
}
```

6 Loose coupling

The implementation of `Particle` contians hard-wired use of OpenGL-specific code. If we wanted to rewrite our program to use a different graphics backend, we would have to change all the client code that relates to graphics. `Particle` is tightly coupled to the graphics system.

This might not seem like such a big deal in our little program, but as the code grows and we introduce more graphics code into it, it will become a much greater problem.

We can loosen the coupling between `Particle` and the graphics system, by getting `Particle` to ask `Simulation` to draw a circle of the appropriate colour and size at the appropriate point, and let `Simulation` take care of the details of how that is done. This idea should be starting to get familiar:

- Provide a high-level interface (a function signature which promises to draw a circle).
- Hide the implementation details behind the interface (the exact calls needed to draw circles in OpenGL).

Move all OpenGL-specific code out of `Particle` and hide it behind `Simulation`'s interface.

Notice that, once you have done this, all the OpenGL headers have disappeared from `main.cc` and from the `Pariticle` files. The fact that we happen to be using OpenGL as our graphics engine is now completely hidden behind the inerface of `Simulation`. If we decide to use a different graphics engine, we could rewrite the internals of `Simulation` but nothing outside it would have to be touched.

7 Dynamic dispatch

It's time to introduce more variety into our particles.

At first, we just want to make sure that we can get dynamic dispatch to work, without getting distracted by other considerations, so we will take some design choices which will have to be fixed later.

Write a subclass of `Particle` called `SquareParticle`. It should behave just like `Particle`, except that, when drawn, it looks square rather than circular. Include both types of particle in the simulation.

To keep things simple, for now, we take the following **temporary** shortcuts:

- Making `SquareParticle` a subclass of (Circular) `Particle` is quite a serious design flaw.
- Let the code for `SquareParticle` live in `Particle.hh` and `Particle.cc`.
- Pretend that the 'radius' of a square is half its side length.

You will need to provide a `square` function in `Simulation`, which will take care of the details of drawing squares.

7.1 Dynamic dispatch not working?

- Did you use `virtual` to switch on dynamic dispatch?
- Are you dispatching through a pointer or a reference?
- Are you sure your particle isn't being sliced (copied by value) anywhere?

8 Class hierarchies

There is much to be said about the relationship betwen types in type herarchies, but the following

principle is a decent first-order approximation:

> Subclasses should be specializations of superclasses.

Just because we *can* make `SquareParticle` a subclass of (Circular) `Particle`, doesn't mean that we *should*. Such a relationship violates the above principle, and is likely to lead to confusion and difficulties in programs of significant complexity.

8.1 Abstract classes

> Make an *abstract* `Particle` class with two *concrete* subclasses: `CircularParticle` and `SquareParticle`.

Recall that an abstract class is one with at least one *purely virtual* member function (one with no implementation) and that the syntax for creating purely virtual member functions is

```
virtual return_type function_name(/* parameters */) = 0;
```

To avoid excessive editing effort, leave the code for `SquareParticle` and `CircularParticle` in `Particle.hh` and `Particle.cc` for the time being.

8.2 Tidying up

Looking at our code, we should be uncomfortable about the fact that our square particles have a radius. You might also like to think about how you might introduce new types of particles for which a radius is not even remotely sensible.

> Remove the radius datum from `Particle`. Rewrite `Particle::move` so that it will work for any new kinds of particles we might care to invent.

To keep this exercise sufficiently simple, assume that no particles will ever spin: they simply bounce off

the boundary, preserving their orientation, as soon as they touch the boundary.

8.3 TODO Extensibility

Discuss how new kinds of particles can be added by the clients, very easily. Maybe even mention the *open-closed principle*.

9 Aggregation vs. Inheritance

Adding further new kinds of particles such as triangles or rectangles would be straightforward now. (Would you be tempted to make rectangle a subclass of square, or maybe square a subclass of rectangle?)

But what if we wanted to add new kinds of particles in an orthogonal direction? All our particles move in straight lines: what if we wanted to add particles which follow circular paths, and zig-zag paths etc.?

One approach would be to have one class for each combination:

LinearCircle	LinearSquare
CircularCircle	CircularSquare
ZigZagCircle	ZigZagCircle

This would lead to a class explosion: For N different shapes, and M different motions, you would need to write $N \times M$ classes.

What is more, you would need to duplicate the square-drawing code in each of the square drawing classes, and so on. (Though this could be mitigated with multiple inheritance or passing function pointers or functors, the class explosion would still be very undesirable.)

An alternative approach is the following

- Create a hierarchy of classes whose only purpose is to draw different shapes
- Create a separate hierarchy whose only purpose is to move particles along different paths.
- Have a single Particle type, which can be instantiated with different combinations of members of the above hierarchies, as parameters.

In this way we will need to implement only $N + M$, rather than $N \times M$ classes.

9.1 A motion strategy

To avoid getting overwhelmed, let's take it in stages. The first step is to take

- Create an abstract class `Motion`, with concrete subclass `Linear`.

- Give `Particle` a `Motion*` member.
- Move the body of `Particle::move` to `Linear::move`, and let the fromer forward the request to the latter, via the `Motion` member.
- Make the `Linear` class a `friend` of `Particle`.

Out in the wild you will hear a lot of nonsense about friends. Just like in life, in C++ it is perfectly OK to have friends, as long as you pick your friends carefully. We'll discuss this in greater depth when we have finished the whole transfromation.

Notice that what we have done amounts to using a different mechanism to enable polymorphic motion. The straight froward way of allowing particles to exhibit different kinds of motion would be to make, `Particle::move` a virtual function, which enables subclasses to provide different kinds of motion by overriding it. Now, `Particle::move` is a nonvirtual function, but it forwards the request to `motion->move(...)`: this enables us to change the motion behaviour of the particle, not by changing the particle's class, but by injecting a different motion strategy into the particle object.

In short, the subclasses of `Motion` act as the different implementations of `Particle::move`.

Our new way of doing it offers some significant advantages in terms of flexibility:

- The motion strategy can be changed at run time.
- The motion strategy can be varied independently of other behaviours within the class.

This idea forms the basis of the GoF *Strategy* and *Command* Patterns.

9.1.1 Introducing variety

Now that we have the basic mechanism in place, let's make use of it, by introducing different kinds of motion.

Implement two new subclasses of `Motion`: `RCircular` and `LCircular`, which move the particles in circles, turning in opposite directions.

The interaction between the particle and the boundaries of our universe is independent of the motion strategy. How are you going to avoid copy-past code reuse?

Mouse over to see answer.

Notice that `Motion` has to make new friends.

9.1.2 Friends aren't evil

The nonsense you are likely to hear about friends often goes something like this:

> Friends can access private members from outside the class, which breaks encaplusation, and makes your programs not object-oriented which means that we're all going to die.

Such arguments betray a fundamental failure to understand what object oriented programming is about: it is not about copious use of the `class` and `private` keywords; it is about high-level interfaces allowing you to be unaware of low-level details. Please remember that it is perfectly possible to write beautifully object-oriented programs in languages which do not have `class` or `private`.

In our example above, `Particle` has a number of `friend` classes. What is the purpose of these classes? They act as a Strategy pattern implementation of the member function `move`. There is no good reason for preventing that function from accessing private members of `Particle` when it is implemented as a Strategy, while allowing the same function that access when it is implemented as a simple member function.

In short, `Motion` and its subclasses should have access to the private members of `Particle`, because `Motion` and its subclasses are, essentially, implementations of a member function of `Particle`.

> Friendship does **not** break encapsulation: it merely changes the shape of the capsule.

Note that the author of `Particle` decides who `Particle`'s friends are. By declaring some component to be a friend of a class, you are making it part of the interface of that class.

9.1.3 Alternatives to friendship

We could have implemented the motion strategy without friendship.

1. By providing all sorts of extra accessor functions in the interface of `Particle`.
2. By passing the position and velocity vectors of `Particle` into the strategy.

Those who dislike friends will consider the above to be better approaches to the problem. Arguably, they

are considerably worse, because these approachees actually **weaken** encaplusation, by giving hints about the internal organization of the particles. This is quite a difficult point to grasp if, as is common amongst C++ and Java programmers, you conclude that *encapsulate* means "use `private` to forcibly prevent access to data members". If you think that this is a sensible definition of *encapsulation* then you will miss the point, and will be likely to hold some very unhelpful (though widely-believed) opinions.

For example, you might conclude that

- Having lots of low-level getters and setters which betray the inner organization of your class is better than having a single high-level `friend`.
- It is impossible to do object-oriented programming in Python.

This is just nonsensical, and highly counterproductive.

9.1.4 Encaplusation

> Encapsulation: providing the ability to use an object whithout knowing its internal details.

In Java and C++ circles, it is widely believed that the capluse must be locked, and that this is the point of encapsulation. To become a good object-oriented programmer you must understand that the capsule is there for your convenience, it is not there to provide some dogmatic rules for people to follow blindly.

9.1.5 Why not using friendship might be better

As it stands, clients cannot add new motion stategies. Why?

Mouse over to see answer.

If implemented using friendship, the `Motion` hierarchy is not open for extension (ToDo: find a good place to talk about the open-closed principle.): the `Motion` hierarchy forms part of the interface of `Particle` and therefore straddles the boundary of the capsule. As such, it makes sense that clients should not be able to mess with it.

If the `Motion` strategy is implemented using one of the alternatives suggested above, then it is moved

entirely outside the capsule. Clients will now be able extend this hierarchy, but this comes at the cost of making the `Particle` interface more complicated and having it betray some internal details: it doesn't matter that the data members are private; their corresponding accessors guide clients to awareness of the internal organization.

9.1.6 So, what shall I do?

As ever, you should strive for

- loose coupling between components
- generality and flexibility

So, how about

```
class Motion {
    virtual void move(Vector& position, Vector& velocity, double dt) = 0;
};
```

Like this, the motion strategy can be used with *anything* that is prepared to provide it with a position vector, a speed vector and a `dt`. Such a motion strategy needs no knowledge of its clients at all.

So, in this case, I would probably go for the loosely coupled approach outlined just above. But do remember that our original motivation for writing the Motion strategy was to make an exchangeable implementation of a **method** of the `Particle` hierarchy. Viewed as a method of the `Particle` hierarchy

- It is reasonable to have it strongly coupled to the `Particle` hierarchy.
- It makes no sense to have it apply to a broader set of types.
- It is reasonable to allow it access to `Particle`'s private members.

Don't conclude that the approach which best fits our specific example is also the best approach in all other situations.

9.2 A shape strategy

Let's take a step back and look at what we have achieved, by writing the `Motion` strategy. Let us say that we want our particles to have the following possible shapes:

- Circle
- Square
- Rectangle
- Triangle

- Star

and the following types of motion

- Linear
- Right Circular
- Left Circular
- Zig Zag
- Random Walk

With motion implemented as a strategy, we can do this with 5 concrete classes for the shapes, and 5 concrete classes for the motions. Had motion been implemented as a simple virtual function, we would need 25 concrete classes: one for each shape-motion combination.

As it stands, `Particle::draw` is a purely virtual function, and the subclasses provide implementations which distinguish between particles of different shapes. We could implement it as a strategy, just like we did in the case of `move`. Notice that the `Motion` instances are stateless: they contain no data, their only purpose is to provide a means of injectng a function into `Particle` instances. One consequence of this is that we need just a single instance of each strategy. If we were to implement a `Shape` strategy, its instances would not be stateless: they would have to hold information such as the radius of a circle, the height and width of a rectangle. Consequently, each particle would need to have a separate instance of its shape strategy.

But, now that `move` is a strategy, which already gives us the ability to vary shape independently of motion, we wouldn't gain anyhing by making shape a strategy too.

Or would we?

9.3 State pattern

Notice that, because motion is a strategy, it would be fairly easy to change a particle's motion at run time: simply make the internal `motion` pointer point to a different strategy. Changing the particle's shape at run time is, currently, impossible. The shape is dependent on the particle's type, and an object's type is fixed for its lifetime, in C++.

So, rewriting our particles in terms of a shape strategy would give us the added ability to change a given particle's shape at run time.

Recall the way in which we described the situation earlier: "The shape is dependent on the particle's type, and an object's type is fixed for its lifetime." And yet, if we create a shape strategy, we are effectively changing the particle's type!

The *State Pattern* is another of the *GoF* design patterns, and in terms of implementation details it is almost identical to *Strategy*. The difference is in the intent you convey when you choose to use one of

those words as opposed to the other. You would prefer to use the word *State* if what you are doing is

- Changing an object's type at run time
- Implementing state machines

while you would prefer to call it a *Strategy* when

- Providing alternative ways of performing some task
- TBD etc.

9.4 Summary / Things that need to be said

9.4.1 Containers and dynamic dispatch

If you are going to dynamically dispatch on a collection of objects stored in a container, it must be a container of pointers to those objects. Remember, dynamic dispatch does not work through values, only pointers or references. Containers cannot contain references.

9.4.2 TODO `dynamic_cast`

9.4.3 Inheritance vs Aggregation

- Inheritance is necessary for dynamic dispatch.
- Inheritance is used to implement dynamic polymorphism.
- Aggregation is an alternative mechanism for implementing dynamic polymorphism, and it is more flexible than inheritance.
- The *Strategy* and *State* patterns can be used to implement class or behaviour switching at runtime.
- The *Strategy* pattern helps prevent class explosion.
- Inheritance is necessary for the implementation of the *Strategy* and *State* patterns.

9.4.4 Is OOP useful?

Object oriented techniques can make your sofware cleaner, more flexible, more maintainable etc. They can also have the opposite effect. Which effect prevails depends on the skill and judgement used in their applicaiton. These skills develop with practise and experience.

9.5 TODO Purely Abstract class

9.6 TODO `dynamic_cast`

Validate