# Iterating

In order to implement your actual Mechanic logic you have to be able to process all of the Filter-matching Subjects (and Subjectives, which are in turn - Subjects). For the purpose of effectiveness (and consistency) this is mainly done through Chains. You iterate those Chains by iterating through all the Subjects and Subjectives inside them.

Chains are iterated with the help of Cursors. Those are very much like container iterators.

## C++ Workflow

Iterating a Chain is done through a special type of object called *Cursor*. You can have as many of those, but usually one is enough:

```
FChain::FCursor Cursor = Chain->Iterate();
```

The solid-variant would logically be like so:

```
FSolidChain::FCursor SolidCursor = SolidChain->Iterate();
```

When you got the Cursor needed, you can construct a simple while loop like so:

```
while (Cursor.Provide())
{
  auto Trait = Cursor.GetTrait<FMyTrait>();
  ...
}
```

The `Provide()` method prepares the needed state and would return `false` when there are no more slots available (`true`, otherwise).

With a solid Cursor you can also get a direct (copy-free) reference to a Trait (with a `GetTraitRef()` method):

```
while (SolidCursor.Provide())
{
  auto& Trait = SolidCursor.GetTraitRef<FMyTrait>();
  ...
}
```

Please note, that the Chain gets disposed automatically when all of the iterating Cursors have finished providing (iterating) the slots. To suppress this behavior use explicit `Retain()`/`Release()` calls for a custom lifetime organization:

```
Chain->Retain(); // Grab the chain.
FChain::FCursor Cursor = Chain->Iterate();
while (Cursor.Provide())
{
  ...
}
// Do some other stuff with the chain.
// It's now guaranteed to not be disposed.
...
Chain->Release(); // Free the chain.
```

## Embedded Cursors

Apparatus provides a way to iterate the chains via embedded (self-allocated) cursors. This is mainly utilized internally, by the Blueprints and generally should be avoided within your C++ code.

Your basic loop is quite simple. It's just a `while` statement with a single condition:

```
while (Chain.BeginOrAdvance())
{
  ...
}
```

Inside this loop you can implement some actual logic. Using the Subject's direct or Chain's utility methods:

```
while (Chain.BeginOrAdvance())
{
  FSubjectHandle Subject = Chain.GetSubject();
  UMyDetail* MyPosition = Chain.GetDetail<UMyDetail>();
  FMyTrait   MyVelocity;
  Chain.GetTrait(MyVelocity);
  MyPosition->X += MyVelocity.VelocityX * DeltaTime;
  MyPosition->Y += MyVelocity.VelocityY * DeltaTime;
  ...
  MyVelocity.VelocityX = 0;
  MyVelocity.VelocityY = 0;
  Subject.SetTrait(MyVelocity);
}
```

When the Chain Cursor is gone past the last available Subject/Subjective the Chain becomes disposed and the previously locked Chunks and Belts become unlocked again, applying all the pending changes (if there are any).

## Direct Iterating

If you want to iterate the Chunks directly you can utilize the Chunks Proxies. You will basically need to iterate all of the Proxies collected through the corresponding Enchain method and within each Chunk iterate all of its Subjects-containing Slots.

An example is as such:

```
TArray<TChunkProxy<FSolidSubjectHandle, FJumpingTrait>> ChunkProxies;
Mechanism->Enchain(TFilter<FJumpingTrait>(), ChunkProxies);
for (int32 i = 0; i < ChunkProxies.Num(); ++i) // Iterate through all of the
matching Chunks...
{
    auto& ChunkProxy = ChunkProxies[i];
    for (int32 j = 0; j < ChunkProxy.Num(); ++j) // Iterate through all of
the Chunk's Slots...
    {
        // Perform the necessary logic...
        ChunkProxy.TraitRefAt<FJumpingTrait>(j).Position +=
FVector::UpVector * DeltaTime;
    }
}
```

Please note however, that this approach is totally manual, and doesn't perform any Iterating-time logic checks like the Flagmark-matching.

If you're also performing some topology-changing logic within your loops your Subjects can change their Chunks and Slots in an arbitrary way, so perhaps you also have to be checking for Slots to be (non-)stale.

You would do it like so:

```
for (int32 j = 0; j < ChunkProxy.Num(); ++j) // Iterate through all of the
Chunk's Slots...
{
    if (ChunkProxy.IsStaleAt(j)) continue; // Skip the Subject if it's
either moved or despawned...
    ChunkProxy.SubjectAt(j).SetTrait(FSpeedBoostTrait{10.0f});
}
```

Iterating the Chunks directly (via Proxies) can introduce a certain performance boost as compared to normal iterating and Operating. That's mainly due to being able to control every aspect of the Iterating by hand and exclude all of the checks unnecessary for your logic. You just have to know

exactly what you’re doing and trying to achieve as this way is a bit less secure.