SERIALIZATION (ENCODE/DECODE)

When a structure is sent over the network or written to disk, it is encoded into a string of bytes. Serializable structures have encode and decode methods that write and read from `bufferlist` objects representing byte strings.

ADDING A FIELD TO A STRUCTURE

You can see examples of this all over the Ceph code, but here's an example:

```
class AcmeClass
{
    int member1;
    std::string member2;

    void encode(bufferlist &bl)
    {
        ENCODE_START(1, 1, bl);
        ::encode(member1, bl);
        ::encode(member2, bl);
        ENCODE_FINISH(bl);
    }

    void decode(bufferlist::iterator &bl)
    {
        DECODE_START(1, bl);
        ::decode(member1, bl);
        ::decode(member2, bl);
        DECODE_FINISH(bl);
    }
};
```

The `ENCODE_START` macro writes a header that specifies a *version* and a *compat_version* (both initially 1). The message version is incremented whenever a change is made to the encoding. The compat_version is incremented only if the change will break existing decoders – decoders are tolerant of trailing bytes, so changes that add fields at the end of the structure do not require incrementing compat_version.

The `DECODE_START` macro takes an argument specifying the most recent message version that the code can handle. This is compared with the compat_version encoded in the message, and if the message is too new then an exception will be thrown. Because changes to compat_verison are rare, this isn't usually something to worry about when adding fields.

In practice, changes to encoding usually involve simply adding the desired fields at the end of the encode and decode functions, and incrementing the versions in `ENCODE_START` and `DECODE_START`. For example, here's how to add a third field to `AcmeClass`:

```
class AcmeClass
{
    int member1;
    std::string member2;
    std::vector<std::string> member3;

    void encode(bufferlist &bl)
    {
        ENCODE_START(2, 1, bl);
        ::encode(member1, bl);
        ::encode(member2, bl);
        ::encode(member3, bl);
        ENCODE_FINISH(bl);
    }

    void decode(bufferlist::iterator &bl)
    {
        DECODE_START(2, bl);
        ::decode(member1, bl);
        ::decode(member2, bl);
        if (struct_v >= 2) {
            ::decode(member3, bl);
```

```
        }
        DECODE_FINISH(bl);
    }
};
```

Note that the compat_version did not change because the encoded message will still be decodable by versions of the code that only understand version 1 – they will just ignore the trailing bytes where we encode `member3`.

In the decode function, decoding the new field is conditional: this is because we might still be passed older-versioned messages that do not have the field. The `struct_v` variable is a local set by the `DECODE_START` macro.