```
// filter shortcuts - derived from zolzer, chamberlin, stinson and tarrabia

/* the simplest one pole
this is typically used for parameter smoothing.
if c is near 1.0, the out slowly moves to the new value (lowpass)
if c is near 0.0, the out follows the new value closely (passthru)
out = new * (1 - c) + out * c;

c can be calculated for a given frequency by
exp(-2.f*PI*cutoff/samplingRate);

get a highpass by subtracting in from out -
*/

// moving forward
/*
 most filters have a section of code for setting the frequency, resonance and
gain (calculating coefficients) and another section for filtering the audio
(using those coefficients and saving internal state/delays)

it’s typical for the frequency/resonance to coefficient calculation to be
more cpu intensive than the filtering of sound
*/
// first and second order filters derived from udo zolzer DAFX book
/*
 an allpass filter does not change the amplitude, but shifts the phase 180 at
nyquist
adding the original signal to the output of this allpass results in a lowpass
*/

// first order allpass

void foAllpass::setFrequency(float f)
{
      tf = tan(pi * f/sampleRate);
      c = (tf -1.0f)/(tf + 1.0f);
}

float foAllpass::process(float in, float c)
{
      out = c * in + in1 - c * out;
      in1 = in;
      return(out);
}

// lowpass requires one more line of code

float foLowpass::process(float in, float c)
{
      out = c * in + in1 - c * out; // out is allpass
      out = in + out;
      in1 = in;
      return(out);
}
// highpass requires inverting the phaseshifted out

float foLowpass::process(float in, float c)
{
```

```
	out = c * in + in1 - c * out; // out is allpass
	out = in - out;
	in1 = in;
	return(out);
}

/*
a second order allpass shifts the phase 360 at nyquist, so it is 180
( inverted) somewhere in between
 second order allpass + input gives you band reject,
 input - allpass gives you band pass
 two coefficients are required (frequency and bandwidth)
*/

void soAllpass::setFrequency(float f, float bw)
{
	tf = tan(pi * bw/sampleRate);
	c = (tf -1.0f)/(tf + 1.0f);
	d = -cos(twopi * f/sampleRate);
}

float soAllpass::process(float in, float c)
{
	out = -c * in + d * (1.0f - c) * in1 + in2
	-d * (1.0 - c) * out + c * out1;

	out1 = out;
	in2 = in1;
	in1 = in;
	return(out);
}

// like the first order example - bandpass and reject derived from allpass
float soBandpass::process(float in, float c)
{
	out = -c * in + d * (1.0f - c) * in1 + in2
	-d * (1.0 - c) * out + c * out1;

	out1 = out;
	in2 = in1;
	in1 = in;
	out = in - out;
	return(out);
}

float soBandreject::process(float in, float c)
{
	out = -c * in + d * (1.0f - c) * in1 + in2
	-d * (1.0 - c) * out + c * out1;

	out1 = out;
	in2 = in1;
	in1 = in;
	out = in + out;
	return(out);
}
```

```
/*
 at this point we have reached the complexity of a biquad -
 and there are many cookbook examples of how to compute
 the biquad coefficients.
 here is one from patrice tarrabia from musicdsp for a butterworth lowpass
*/

BQLowpass::BQLowpass()
{
    A1 = A2 = A3 = 0.0;
    B1 = B2 = 0.0;
    X1 = X2 = Y1 = Y2 = 0.0;
    Pi = 4.0 * atan(1.0);
}

BQLowpass::~BQLowpass()
{}

BQLowpass::SetFilter(float frequency, float resonance)
{
    float C, f0;

    if(resonance > sqrt(2.0))
            resonance = sqrt(2.0);
    if(resonance < 0.1)
            resonance = 0.1;
    if(frequency > sampleRate * 0.5)
            frequency = sampleRate * 0.5;

    f0 = frequency/sampleRate;
    if(f0 < 0.1)
            C = 1.0 / (f0 * Pi);
    else
            C = tan((0.5 - f0) * Pi);

    A1 = 1.0 / ( 1.0 + resonance * C + C * C);
    A2 = 2.0 * A1;
    A3 = A1;
    B1 = 2.0 * ( 1.0 - C * C) * A1;
    B2 = ( 1.0 - resonance * C + C * C) * A1;
}

BQLowpass::FilterBlock(float *input, float *output, long samplePerBlock)
{
    long sample;

    for(sample = 0; sample < samplesPerBlock; sample++)
    {
            *(output + sample) = A1 * *(input + sample)
                  + A2 * X1 + A3 * X2 - B1 * Y1 - B2 * Y2;

            X2 = X1;
            X1 = *(input + sample);
            Y2 = Y1;
            Y1 = *(output + sample);
    }
}
```

```
/*
the moog filter concatenates 4 single pole filters. these single pole filters
are similar to the first order lowpass we derived from the allpass. in
addition, there is a feedback around all 4 filters to add resonance.
this is tim stilson's code from ccrma
*/

/*
a gain table is used so that resonance breaks into oscillation similarly for
all frequencies
*/

static float gaintable[199] = { 0.999969, 0.990082, 0.980347, 0.970764,
0.961304, 0.951996, 0.94281, 0.933777, 0.924866, 0.916077, 0.90741, 0.898865,
0.890442, 0.882141 , 0.873962, 0.865906, 0.857941, 0.850067, 0.842346,
0.834686, 0.827148, 0.819733, 0.812378, 0.805145, 0.798004, 0.790955,
0.783997, 0.77713, 0.770355, 0.763672, 0.75708 , 0.75058, 0.744141, 0.737793,
0.731537, 0.725342, 0.719238, 0.713196, 0.707245, 0.701355, 0.695557,
0.689819, 0.684174, 0.678558, 0.673035, 0.667572, 0.66217, 0.65686, 0.651581,
0.646393, 0.641235, 0.636169, 0.631134, 0.62619, 0.621277, 0.616425,
0.611633, 0.606903, 0.602234, 0.597626, 0.593048, 0.588531, 0.584045,
0.579651, 0.575287 , 0.570953, 0.566681, 0.562469, 0.558289, 0.554169,
0.550079, 0.546051, 0.542053, 0.538116, 0.53421, 0.530334,0.52652, 0.522736,
0.518982, 0.515289, 0.511627, 0.507996 , 0.504425, 0.500885, 0.497375,
0.493896, 0.490448, 0.487061, 0.483704, 0.480377, 0.477081, 0.473816,
0.470581, 0.467377, 0.464203, 0.46109, 0.457977, 0.454926, 0.451874,
0.448883, 0.445892, 0.442932, 0.440033, 0.437134, 0.434265, 0.431427,
0.428619, 0.425842, 0.423096, 0.42038, 0.417664, 0.415009, 0.412354,
0.409729, 0.407135, 0.404572, 0.402008, 0.399506, 0.397003, 0.394501,
0.392059, 0.389618, 0.387207, 0.384827, 0.382477, 0.380127, 0.377808,
0.375488, 0.37323, 0.370972, 0.368713, 0.366516, 0.364319, 0.362122,
0.359985, 0.357849, 0.355713, 0.353607, 0.351532, 0.349457, 0.347412,
0.345398, 0.343384, 0.34137, 0.339417, 0.337463, 0.33551, 0.333588, 0.331665,
0.329773, 0.327911, 0.32605, 0.324188, 0.322357, 0.320557, 0.318756,
0.316986, 0.315216, 0.313446, 0.311707, 0.309998, 0.308289, 0.30658,
0.304901, 0.303223, 0.301575, 0.299927, 0.298309, 0.296692, 0.295074,
0.293488, 0.291931, 0.290375, 0.288818, 0.287262, 0.285736, 0.284241,
0.282715, 0.28125, 0.279755, 0.27829, 0.276825, 0.275391, 0.273956, 0.272552,
0.271118, 0.269745, 0.268341, 0.266968, 0.265594, 0.264252, 0.262909,
0.261566, 0.260223, 0.258911, 0.257599, 0.256317, 0.255035, 0.25375 };

void moogFilter::setFrequency(float freq, float reson)
{
       f = 2.0 * f freq/sampleRate;

       //code for setting pole coefficient based on frequency
       moogP = -0.69346*f*f*f - 0.59515*f*f + 3.2937*f - 1.0072;
       //cubic fit by DFL, not 100% accurate but better than nothing...

       //code for setting Q
       index = moogP * 99;
       indexInt = floor(index);
       indexFrac = index - indexInt;
       moogQ = reson *
            ((1.0 - indexFrac) * gaintable[indexInt + 99]
     + indexFrac * gaintable[indexInt + 100]) * 1.2f;

}
```

```
float moogFilter::process(float in)
{

        out = 0.25f * ( in - out ); //negative feedback
        for(int pole = 0; pole < 4; pole++)
        {
             float temp;
             temp = state[pole];
             out = saturate( out + moogP * (out - temp));
             state[pole] = out;
             out = saturate( out + temp );
        }
}

/*
a state variable filter has two integrators feeding back into each other, and
a network that derives all 4 filter types simultaneously
this is derived from the BASIC code in hal chamberlain's musical applications
of microprocessors. damping and freqCoef are limited to keep filter more
stable
*/

SVFilter::ProcessBlock(float *input, float *output, long samplesPerBlock,
float frequency, float resonance, float drive)
{
        long sample;
        float in, out;
        float freqCoef, damping;

        freqCoef = 2.0*sin(pi * (frequency/sampleRate)));
        if(freqCoef > 0.5)
             freqCoef = 0.5;

        damping = 2.0*(1.0 - pow(resonance, 0.25));
        if(damping > 2.0)
             damping = 2.0;
        if(damping > (2.0/freqCoef - (freqCoef * 0.5)))
             damping = 2.0/freqCoef - (freqCoef * 0.5);

        for (sample=0; i<samplesPerBlock; sample++)
        {
          in    = *(input+i);

          notchOut = in - (damp * bandOut);
          lowOut   = lowOut + (freqCoef * bandOut);
          highOut  = notchOut - lowOut;
          bandOut  = bandOut + (freqCoef * highOut);

          output[i] = lowOut;
        }
}
```