Filter

From Crypto++ Wiki

A Crypto++ filter is the means by which data is transformed in the pipelining paradigm. It is derived from a `BufferedTransformation`. The filter class is the starting point for custom filters in Crypto++. A more advanced examinations can be found at Filter (Intermediate).

Contents

Constructors

```
Filter (BufferedTransformation *attachment=NULL)
```

All classes derived from filter, as well as the filter class itself, take a parameter of the form `BufferedTransformation*`. This parameter specifies a pointer to any object derived from `BufferedTransformation` that is to be attached to the object that is being constructed.

A pointer to an attached object becomes owned by the attached-to object as soon as the attachment relationship commences. The attached object will automatically be freed when the attached-to object is destructed.

If no object-to-be-attached is specified when constructing an object derived from filter, or if a null pointer is passed, an object of a predetermined type will be created. Currently, the default is to create a `MessageQueue` object. A `MessageQueue` will store all data it receives through `Put` methods for later retrieval via `Get` methods, in the same order in which data was received.

Methods

Attach

Attaches another transformation object to the end of the current attachment chain.

Detach

Deletes the current attachment chain, destructing and freeing all objects it consists of; then, it attaches the specified object. If a null pointer is passed, an attachment object of the default type is created.

Attachable

To distinguish between a filter and a `BufferedTransformation` object, call its `Attachable` method. If the object is a filter, the method will return true; if it is a `BufferedTransformation`, it will return false.

AttachedTransformation

Returns a pointer to the currently attached transformation object. Should never be null.

Sample Programs

Source to Sink

The first sample is very basic The contents of the first string are placed in second string. In this case, there is no filter in the pipeline. Note that the source (*s1*) is not drained. That is, *s1* will still have the value of 'Filter'.

```
string s1 = "Filter", s2;
StringSource(s1, new StringSink(s2));

cout << s2 << endl;
```

Output: **`Filter`**

ArraySource and HexEncoder

The second sample adds a `ArraySource` and `HexEncoder` to the pipeline. In versions prior to Crypto++ 5.6, use a `StringSource` instead of an `ArraySource`.

```
byte data[] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
string encoded;

ArraySource(data, sizeof(data), true,
    new HexEncoder(
        new StringSink(encoded)
    )
);
```

Redirector

The third sample uses a `Redirector`. A `Redirector` will end an attachment chain, but still pass the data it

has received to another filter. The behavior is useful when an intermediate result from a filter is required (the filter would otherwise be destroyed, and the result would be lost). Most notable is the `DecodingResult` from a `HashVerificationFilter`. In this situation, we **do not** want the StringSource to own the AuthenticatedDecryptionFilter since the StringSource will destroy the filter. To accomplish the task, a `Redirector` would be used. Notice that the `Redirector` takes a reference to an object, and not a pointer to an object.

```
CCM< AES, TAG_SIZE >::Decryption d;
d.SetKeyWithIV(key, sizeof(key), iv, sizeof(iv));
...

AuthenticatedDecryptionFilter df(d,
    new StringSink(recovered)
); // AuthenticatedDecryptionFilter

// Cipher text includes the MAC tag
StringSource(cipher, true,
    new Redirector(df)
); // StringSource

// If the object does not throw, here's the only
//  opportunity to check the data's integrity
bool b = df.GetLastResult();

if(b)
    cout << recovered << endl;
```

MeterFilter

The fourth sample demonstrates use of a `Redirector` and `MeterFilter` to count the number of bytes output by the `HexEncoder` and subsequently input to the `StringSink`.

```
byte data[] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };

string encoded;
MeterFilter meter(new StringSink(encoded));

ArraySource(data, sizeof(data), true,
    new HexEncoder(
        new Redirector(meter),
        true /*UCase*/, 2 /*Group*/
    )
);

cout << "processed " << meter.GetTotalBytes() << " bytes" << endl;
cout << encoded << endl;
```

Output: **`Processed 23 bytes`**
`00:01:02:03:04:05:06:07`

Recall that the data in the `MeterFilter` would not be available if using a stock pipeline since the meter would be destroyed when the `ArraySource` destructor was invoked. The following code, which will surreptitiously produce correct results most of the time, is **broken**. The pointer is invalid once the line `ArraySource(...);` executes.

```
byte data[] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };

string encoded;
```

```
MeterFilter* meter = NULL;
ArraySource(data, sizeof(data), true,
    meter = new MeterFilter(
        new StringSink(encoded)
    )
);

cout << "processed " << meter->GetTotalBytes() << " bytes" << endl;
```

HexEncoder and Attach

The fifth sample attaches a `HexEncoder` and then encodes the binary data. The binary data is manually input to the filter using the `Put` method. To signal the end of input, `MessageEnd` is called.

To format the data for pretty printing, the code below uses " 0x" as a separator. Because "0x" is a separator, the first byte encoded will lack "0x" (there's no need for a separator for the first byte). To work around the behavior, the sink is assigned "0x" before inputting any data.

```
byte data[] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };

string encoded = "0x";
HexEncoder encoder(NULL, true, 2, " 0x");

encoder.Attach(new StringSink(encoded));

encoder.Put(data, sizeof(data));
encoder.MessageEnd();

cout << encoded << endl;
```

Output: **0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07**

Additional Filters

Please visit Filter (Intermediate) for techniques extending beyond basic, such as custom filters.

Categories: Sample | Filter